

16. IL LINGUAGGIO SQL

Il **linguaggio SQL** (*Structured Query Language*) è un linguaggio che consente di interrogare, gestire e manipolare i dati di un database *relazionale* tramite **comandi** (che prendono il nome di *query*) formati da un insieme di **parole chiave** prese dal linguaggio corrente *inglese*.

Grazie ad una sintassi semplificata, i **comandi** utilizzati hanno una struttura *intuitiva*, *lineare* ed *espressiva*.

Il **linguaggio SQL** è inoltre un linguaggio di **tipo dichiarativo** in quanto è un tipo di linguaggio in cui i **comandi** si limitano a descrivere “cosa” si vuol fare e non “come” esso sarà fatto.

N.B. Il **linguaggio SQL** (*Structured Query Language*) **NON** è un **linguaggio di programmazione** in quanto non è in grado di descrivere i passi elementari di elaborazione attraverso le tipiche strutture di controllo e/o attraverso procedure e/o funzioni.

Il linguaggio SQL può essere usato in:

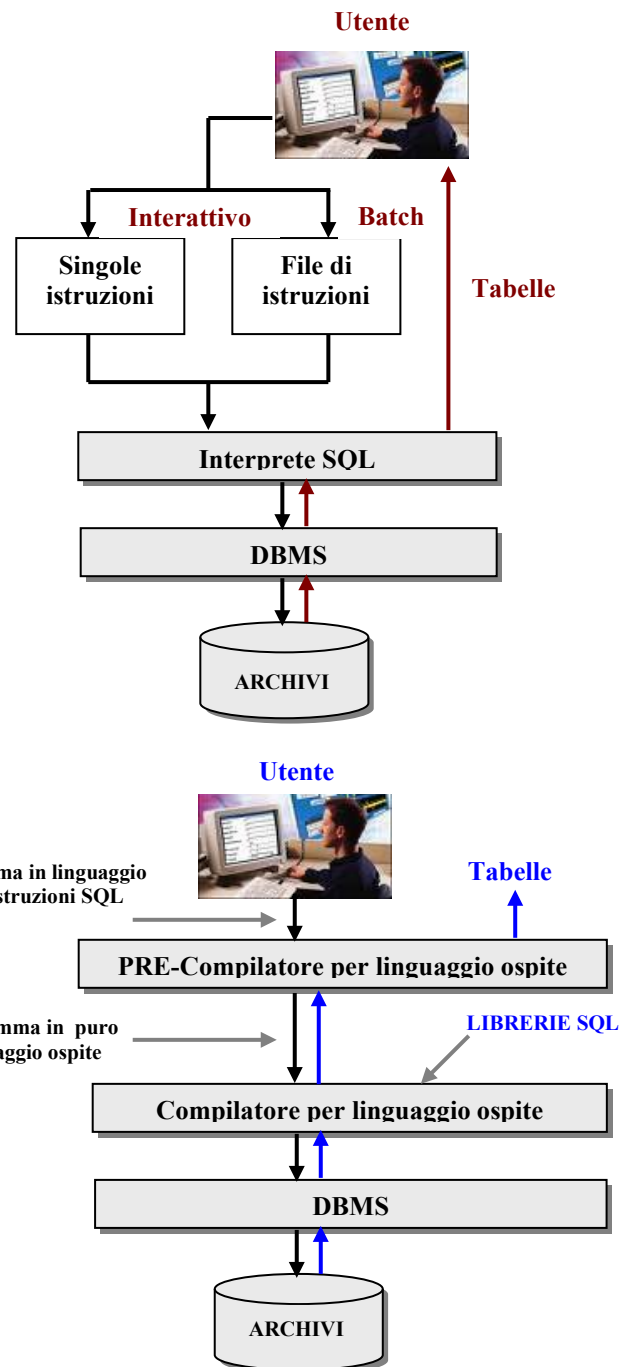
- **modalità stand-alone**: in questa modalità può essere classificato come **query language interattivo**. I **comandi SQL** possono essere inviati al sistema operativo in due modalità (non esclusive):
 - a) in modo **interattivo** utilizzando un’interfaccia grafica con menù, finestre ed icone (**G.U.I**) oppure un’interfaccia a caratteri come una console dei comandi (**C.L.I.**)
 - b) in modo **batch** (creando file di **comandi SQL** da eseguire in gruppo).

In entrambe le modalità verrà invocato l’**interprete SQL**.

Le **interrogazioni** o **query** vengono composte utilizzando le operazioni **dell’algebra relazionale** ottenendo come risultato sempre una **tabella** che rappresenta una relazione.

- **modalità embedded**: in questa modalità è possibile utilizzare **comandi SQL** all’interno di istruzioni di altri linguaggi di programmazione (ad esempio *C*, *Java*, *C++*, *PHP*, *ASP*, etc.) detti “linguaggi ospite”.

In linea di massima un programma scritto in **linguaggio ospite compilato** (che incorpora **comandi SQL** al suo interno) subirà un primo processo di **precompilazione** (che si occupa di tradurre le sole istruzioni SQL in istruzioni in linguaggio ospite compilato) seguito da una vera e propria **compilazione** (che si occupa di tradurre le istruzioni specifiche del linguaggio ospite)



Noi faremo riferimento al linguaggio SQL utilizzato dal DBMS MySql nella sua versione MariaDB 10.1.37 (all'interno del pacchetto XAMPP 5.6.40)

Il linguaggio SQL assolve alle funzioni di:

- **DDL** (*Data Definition Language*) che prevede istruzioni per definire la struttura delle relazioni della base di dati. Serve quindi a creare *tabelle*, *vincoli*, *viste* ed *indici*;
- **DML** (*Data Manipulation Language*) che prevede istruzioni per manipolare i dati contenuti nelle diverse tabelle. Serve in particolare per *inserire*, *cancellare* e *modificare* ennuple;
- **DCL** (*Data Control Language*) che prevede istruzioni per controllare il modo in cui le operazioni vengono eseguite. Consente di creare e cancellare gli utenti, gestire il *controllo degli accessi* a più utenti ed i *permessi* agli utenti autorizzati;
- **QL** (*Query Language*) che permette di interrogare il database per ritrovare le informazioni relative alla chiave di ricerca impostata dall'utente

Il **linguaggio SQL** è “**relazionalmente completo**”, nel senso che ogni espressione dell'algebra relazionale può essere tradotta in SQL.

Il modello dei dati di SQL è basato su **tabelle** anziché **relazioni**.

IDENTIFICATORI E TIPI DI DATI

SQL non è un linguaggio case sensitive: quindi le istruzioni possono essere scritte usando sia i caratteri minuscoli che quelli maiuscoli.

Generalmente le istruzioni terminano con il **punto e virgola ;** (ma non per tutte le versioni).

N.B. E' consigliabile usare i caratteri maiuscoli (anche se non richiesti) per le parole chiave del linguaggio

Gli **identificatori** utilizzati per i nomi di tabelle e di attributi devono:

- avere lunghezza max di 18 caratteri;
- iniziare con una lettera;
- contenere come unico carattere speciale l'underscore ossia ' _ ';

Nella terminologia SQL le **relazioni** sono chiamate *tabelle*; le **ennuple** sono dette *righe* o registrazioni e gli **attributi** sono detti *colonne* delle tabelle

Per riferirsi ad un attributo di una tabella si usa la seguente dot-notation.

<NomeTabella>.<NomeAttributo>

I **tipi di dato** utilizzabili per gli attributi sono riassunti nella tabella consultabile all'url <http://www.riochierego.it/mobile/docs/quinta/lab/tipi-dato-sql.pdf> anche se soccorre precisare che in alcune versioni tali tipi potrebbero essere differenti.

Le **costanti stringa** sono rappresentabili usando indifferentemente i singoli apici (' ') oppure i doppi apici (" ").

Nelle **espressioni** possono anche essere usati i seguenti operatori

- *aritmetici* (+, -, /, *);

n.b. L'operatore modulo può essere rappresentato, se dividendo e divisore sono due interi, come MOD (dividendo, divisore); dividendo % divisore; dividendo MOD divisore;

- *relazionali* (<, >, <=, >=, <>, =);

- *logici* (AND, OR, NOT)

I confronti tra dati numerici sono eseguiti in accordo al loro valore algebrico.

I confronti tra dati alfanumerici sono eseguiti in accordo al valore del corrispondente codice **ASCII** dei caratteri che li compongono cominciando dal carattere più a sinistra

ISTRUZIONI DDL del linguaggio SQL

Creare un nuovo database

Per creare un nuovo database useremo l'istruzione

```
CREATE DATABASE <NomeDB> [AUTHORIZATION <Proprietario>] ;
```

Che crea un nuovo database di nome <NomeDb> ed eventualmente specifica il nome dell'utente proprietario in <Proprietario> ossia dell'utente che possiede i privilegi di accesso e che, come tale, è l'unico a poter svolgere determinate azioni sul database.

Per eliminare un database

```
DROP DATABASE <NomeDB> ;
```

Selezionare un database

Una volta creato un database esso viene aggiunto dal DBMS agli altri database precedentemente creati.

Per impartire comandi SQL per uno specifico database occorre prima selezionarlo utilizzando il comando:

```
USE <NomeDB> ;
```

dove <NomeDb> è il nome di un database creato con l'istruzione CREATE DATABASE.

Mostrare i database di un determinato utente

Per mostrare i database gestibili da un determinato utente, dopo avere effettuato il login con le credenziali assegnate:

```
SHOW DATABASES ;
```

Mostrare le relazioni (tabelle) resenti in un determinato database

Per mostrare le tabelle esistenti in un determinato database:

```
SHOW TABLES ;
```

Descrivere la struttura di una relazione (tabella)

Per descrivere la struttura di una tabella (nome dei campi/attributi, tipo di dati, obbligatorietà/opzionalità del campo, chiave primaria, etc.)

```
DESCRIBE <NomeTabella> ;
```

Creazione della struttura di una tabella e dei vincoli di integrità ad essa relativi

In SQL è possibile creare una nuova tabella con l'istruzione **CREATE TABLE** la cui sintassi è:

```
CREATE TABLE <NomeTabella>
(
  <NomeAttributo1> <Tipo1> [<VincoloSingoloAttributo1>],
  <NomeAttributo2> <Tipo2> [<VincoloSingoloAttributo2>],
  .....
  <NomeAttributoN> <TipoN> [<VincoloSingoloAttributoN>],
  [VincoloGruppoAttributi],
  [<VincoliIntegritàreferenziale>]
);
```

Nella definizione di una tabella, come è possibile vedere, sono presenti i seguenti vincoli:

- relativi ad un **singolo attributo** detti anche “**vincoli di dominio**”;
- relativi ad un **gruppo di attributi** detti anche “**vincoli di ennupla**”;
- di **integrità referenziale**.

a) Vincoli relativi ad un singolo attributo (o "vincoli di dominio"):

Impostano limitazioni da specificare sul valore di un singolo attributo.

Possono essere realizzati attraverso le seguenti clausole:

- **NOT NULL**: indica che il corrispondente attributo non può mai assumere il valore NULL (ciò equivale a dire che tale attributo è **obbligatorio**);
- **DEFAULT** <Valore Default>: assegna all'attributo un valore di default;
- **CHECK** (<Condizione>): serve per specificare un qualsiasi vincolo che riguarda il valore di un attributo che deve verificare una determinata <Condizione>
(Nella <Condizione> posta all'interno della clausola CHECK è possibile usare, oltre agli operatori di confronto, gli operatori IN, NOT IN, BETWEEN...AND, NOT BETWEEN...AND, LIKE <espressione>, NOT LIKE <espressione>)

b) Vincoli relativi ad un gruppo di attributi (o "vincoli di ennupla"):

Impostano limitazioni da specificare sui valori di più attributi.

Possono essere realizzati attraverso le seguenti clausole:

- **PRIMARY KEY** (<Attributo1>,...,<AttributoN>) che indica le colonne facenti parte della chiave primaria specificando un *vincolo di chiave primaria* per il modello relazionale;
- **UNIQUE** (<Attributo1>,...,<AttributoN>) che indica i valori degli attributi specificati (che non formano una chiave primaria) devono essere necessariamente distinti (una chiave candidata);
- **CHECK** (<Condizione>): serve per specificare un qualsiasi vincolo che riguarda il valore di più attributi. (All'interno della clausola CHECK è possibile usare, oltre agli operatori di confronto, gli operatori IN, NOT IN, BETWEEN...AND, NOT BETWEEN...AND, LIKE <espressione>, NOT LIKE <espressione>)

c) **Vincoli di integrità referenziale e politiche di violazione:**

Possono essere realizzati attraverso la seguente tipologia di clausola:

- **FOREIGN KEY** (<Attributo1>,...,<AttributoN>) **REFERENCES** <NomeTabella> (<Attr1>,..., <AttrN>)
[**ON DELETE** | **ON UPDATE**] **NO ACTION** o **RESTRICT** |
CASCADE |
SET NULL |
SET DEFAULT],

Le colonne <Attributo1>,...,<AttributoN> rappresentano la **chiave esterna** e corrispondono alle colonne <Attr1>,...,<AttrN> che formano la **chiave primaria** della tabella <Nome Tabella>.

La parte opzionale successiva è relativa al tipo di politica da seguire, nella tabella che si sta creando, in caso di violazione del vincolo referenziale, durante una operazione di **cancellazione** o **modifica** della ennupla referenziata (ossia la ennupla della tabella <NomeTabella> avente la chiave primaria uguale alla chiave esterna)

Al momento della **cancellazione** (clausola **ON DELETE**) dell'attributo/i chiave interessato/i da un vincolo referenziale, si può avere il seguente comportamento:

- con l'opzione **CASCADE**, vengono cancellate le righe corrispondenti;
- con l'opzione **SET NULL**, vengono impostate a **NULL** le righe corrispondenti;
- con l'opzione **SET DEFAULT**, vengono impostate al valore di default le righe corrispondenti;
- con l'opzione **NO ACTION (analogo a RESTRICT)**, non viene permessa l'esecuzione dell'azione.

(n.b. se la clausola **ON DELETE** non è specificata, **NO ACTION (o RESTRICT)** è l'impostazione di **default**).

Al momento della **modifica** (clausola **ON UPDATE**) dell'attributo/i chiave interessato/i da un vincolo referenziale, si può avere il seguente comportamento:

- con l'opzione **CASCADE**, vengono aggiornate le righe corrispondenti con il nuovo valore;
- con l'opzione **SET NULL**, vengono impostate a **NULL** le righe corrispondenti;
- con l'opzione **SET DEFAULT**, vengono impostate al valore di default le righe corrispondenti;
- con l'opzione **NO ACTION (analogo a RESTRICT)**, non viene permessa l'esecuzione dell'azione.

(n.b. se la clausola **ON UPDATE** non è specificata, **NO ACTION (o RESTRICT)** è l'impostazione di **default**).

Vedi esempio da svolgere (con query)

<http://www.riochierego.it/mobile/docs/quinta/lab/ESERCITAZIONE-db-esempio-DIPENDENTE-REPARTO-SOLO-TESTO-QUERY-ALUNNO.pdf>

Riferendoci alle tabelle presenti nell'esempio sopra indicato, proviamo a dettagliare meglio il significato delle clausole **NO ACTION | RESTRICT**, **CASCADE**, **SET NULL** oppure **SET DEFAULT** nell'ipotesi volessimo in seguito effettuare la cancellazione di una ennupla (**ON DELETE**) oppure l'aggiornamento del valore della chiave di un'ennupla (**ON UPDATE**) referenziata all'interno della tabella che stiamo creando.

Quindi al momento dell'istruzione di **CREATE TABLE** di una certa tabella occorrerà interrogarsi su quale dovrà essere il comportamento più corretto da assumere per garantire l'integrità dei dati presenti nel database, in caso di tale eventualità.

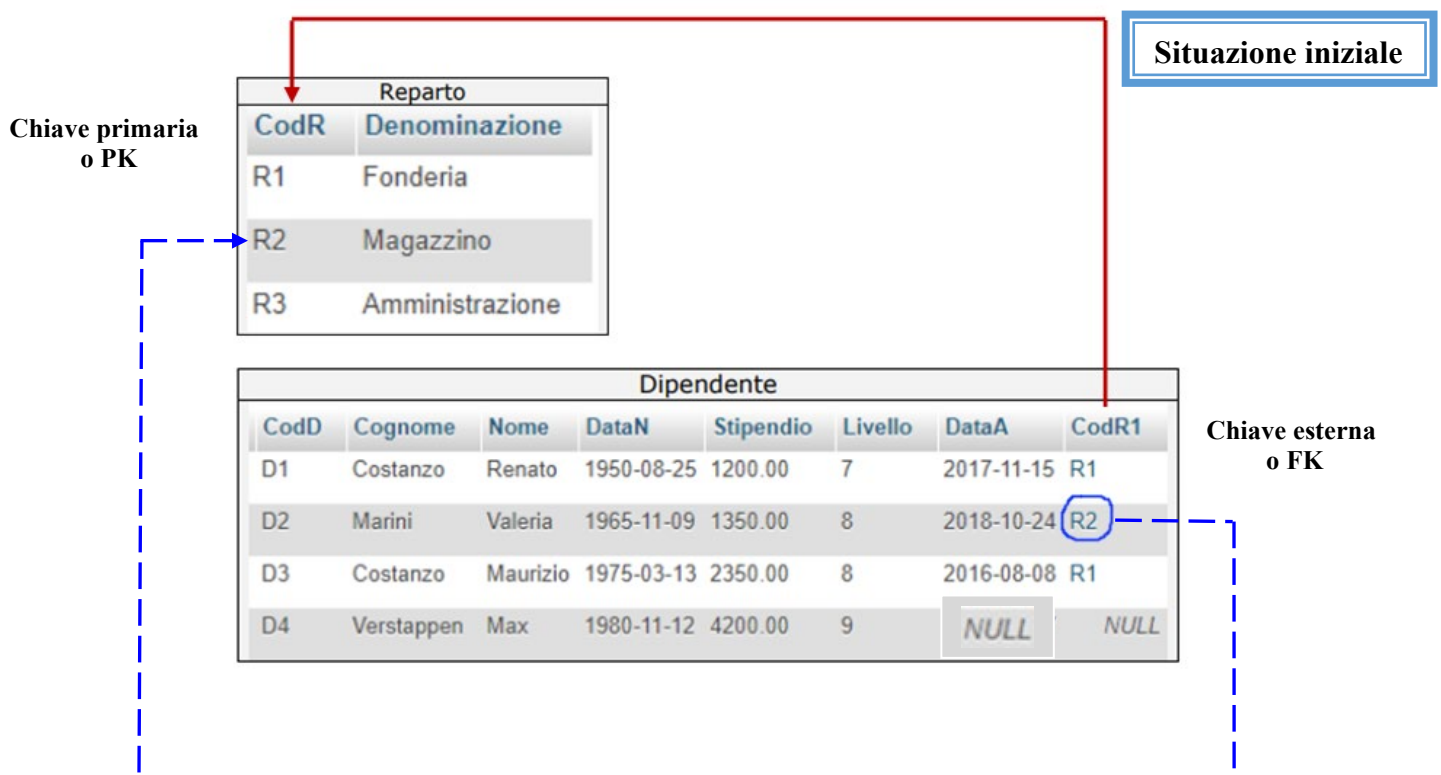
Le domande che ci poniamo sono:

Impedire del tutto l'operazione? Consentirla a cascata? Impostare il campo/i che formano la chiave esterna al valore NULL? Impostare il campo/i che formano la chiave esterna ad un valore di default?

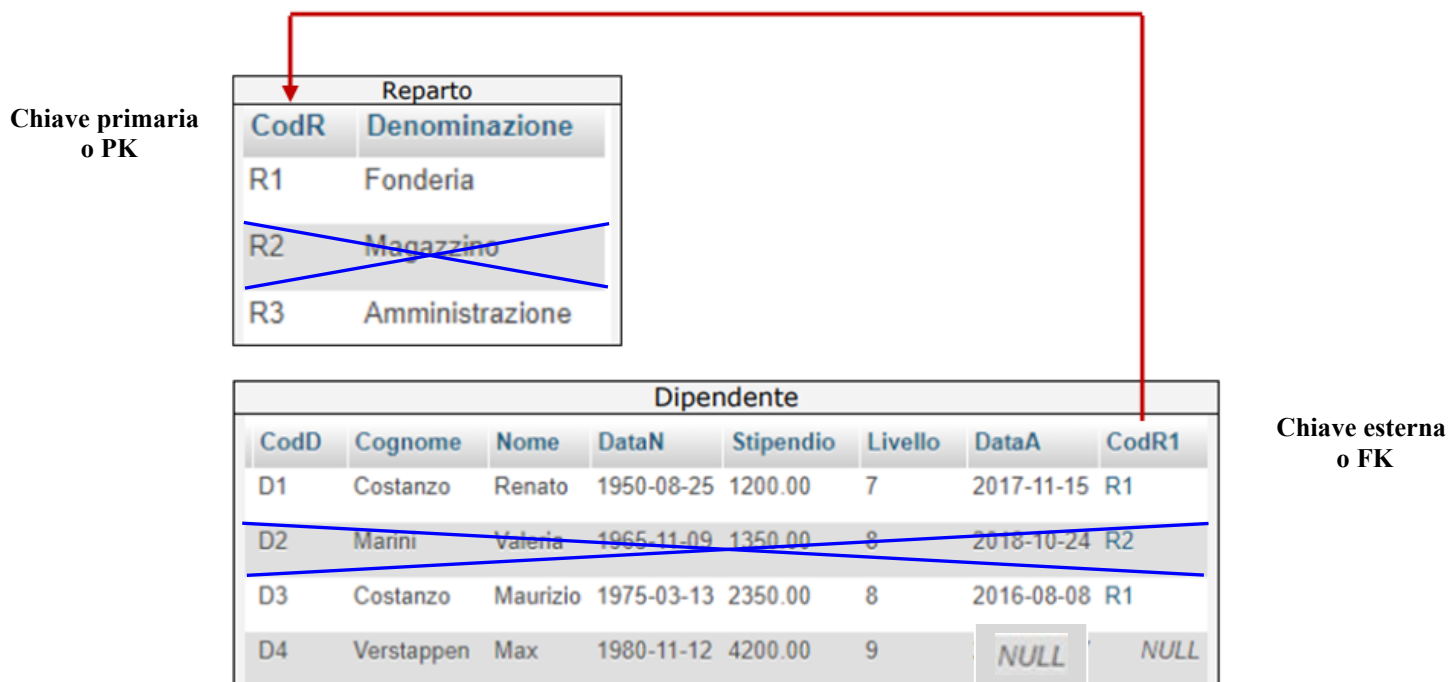
Esempio pratico: Database "AziendaNew"

Cosa accadrà nel nostro database, quando proveremo a **CANCELLARE** (**ON DELETE**) dalla tabella "Reparto" l'istanza avente chiave (primaria) pari a "R2"?

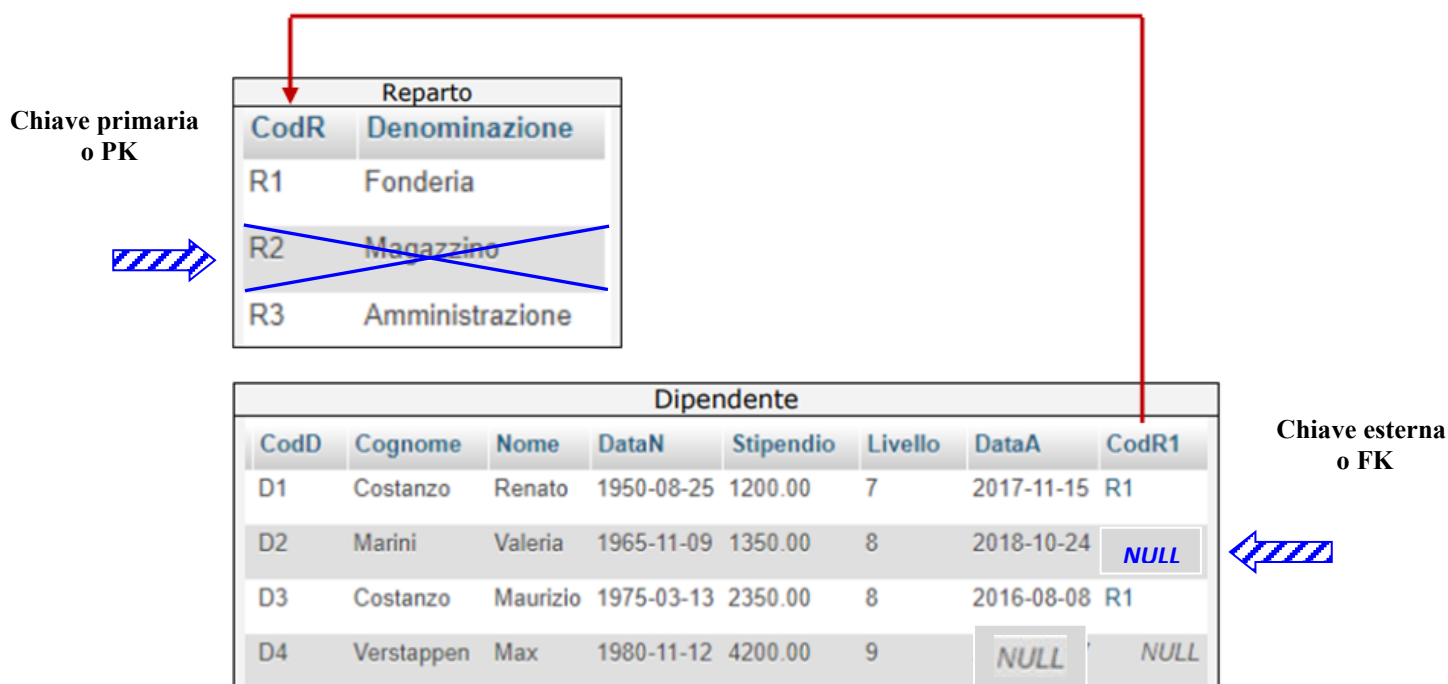
- In caso di **NO ACTION | RESTRICT** (comportamento di default): **operazione di cancellazione NON consentita** sulla tabella "Reparto" perché ciò renderebbe il database incongruente!



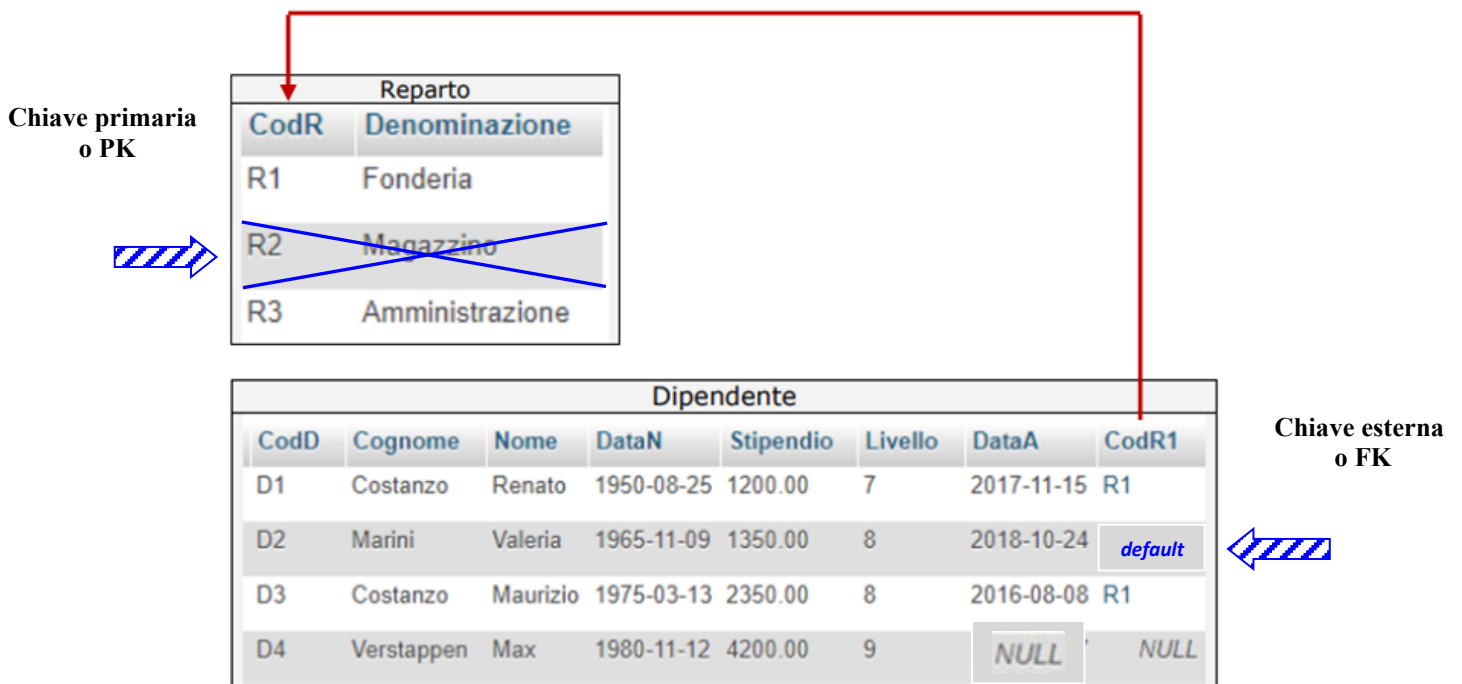
- In caso di **CASCADE**: operazione di cancellazione consentita sulla tabella "Reparto" e, per mantenere il database congruente, verranno cancellate dalla tabella "Dipendente" tutte le istanze che presentano il valore "R2" nel campo "CodR1".



- In caso di **SET NULL**: operazione di cancellazione consentita sulla tabella "Reparto" e, per mantenere il database congruente, nella tabella "Dipendente" verranno poste a NULL (se possibile) tutte le istanze che presentavano inizialmente il valore "R2" nel campo "CodR1".



- In caso di **SET DEFAULT**: **operazione di cancellazione consentita** sulla tabella "Reparto" e per mantenere il database congruente, nella tabella "Dipendente" verranno poste al valore di DEFAULT prestabilito (che deve essere stato GIA' OVVIAMENTE previsto ed impostato), tutte le istanze che presentavano inizialmente il valore "R2" nel campo "CodR1".



Cosa accadrà nel nostro database, quando proveremo ad **AGGIORNARE** (**ON UPDATE**) nella tabella "Reparto" l'istanza avente chiave (primaria) pari a "R2" ponendola, ad esempio, uguale a "R6"?

- In caso di **NO ACTION | RESTRICT** (comportamento di default): **operazione di aggiornamento NON consentita** sulla tabella "Reparto" perché ciò renderebbe il database incongruente!



- In caso di **CASCADE**: **operazione di aggiornamento consentita** sulla tabella "Reparto" e, per mantenere il database congruente, verranno aggiornate nella tabella "Dipendente" tutte le istanze che presentavano il valore "R2" nel campo "CodR1", portandolo a "R6".



- In caso di **SET NULL**: **operazione di aggiornamento consentita** sulla tabella "Reparto" e, per mantenere il database congruente, nella tabella "Dipendente" verranno poste a NULL (se possibile) tutte le istanze che presentavano inizialmente il valore "R2" nel campo "CodR1".



- In caso di **SET DEFAULT**: **operazione di aggiornamento consentita** sulla tabella "Reparto" e per mantenere il database congruente, nella tabella "Dipendente" verranno poste al valore di DEFAULT prestabilito (che deve essere stato GIA' OVVIAMENTE previsto ed impostato), tutte le istanze che presentavano inizialmente il valore "R2" nel campo "CodR1".



Ulteriore esempio da svolgere (con query)

<http://www.riochierego.it/mobile/docs/quinta/lab/ESERCITAZIONE-db-esempio-CON-STORICO-FORNITORE-FORNISCE-ARTICOLO-SOLO-TESTO-QUERY-ALUNNO.pdf>

Vedi esempio già svolto (con query)

<http://www.riochierego.it/mobile/docs/quinta/lab/BIBLIOTECA-Esercizio-Svolto.pdf>

Modificare la struttura di una tabella

Per modificare la struttura di una tabella precedentemente creata nella base dei dati si utilizza l'istruzione **ALTER TABLE** la cui sintassi è:

```
ALTER TABLE <NomeTabella>
    ADD COLUMN      <NomeColonna1> <Tipo1> | <NomeDominio> [eventuali vincoli]
    [FIRST | AFTER <NomeColonna2>] ;
                                oppure
ALTER TABLE <NomeTabella>
    DROP COLUMN     <NomeColonna> ;
                                oppure
ALTER TABLE <NomeTabella>
    ALTER | MODIFY COLUMN <NomeColonna> <NuovoTipoColonna> [eventuali vincoli] ;
N.B. La clausola MODIFY al posto di ALTER per l'istruzione ALTER TABLE è SOLO MySQL
                                oppure
ALTER TABLE <NomeTabella >
    CHANGE <NomeCol_OLD> <NomeCol_NEW> <TipoDati_NEW> [eventuali vincoli NEW] ;
N.B. La clausola CHANGE per l'istruzione ALTER TABLE è SOLO MySQL
```

Dove:

- la clausola **ADD COLUMN** permette di aggiungere alla tabella <NomeTabella> la nuova colonna <NomeColonna1> di <Tipo1> oppure <NomeDominio> eventualmente specificando la posizione dove inserirla ossia all'inizio (con la clausola opzionale **FIRST**) oppure dopo rispetto ad un'altra colonna esistente (con la clausola opzionale **AFTER** <NomeColonna2>);
- la clausola **DROP COLUMN** viene usata per eliminare la colonna <NomeColonna> della tabella <NomeTabella>;
- la clausola **ALTER COLUMN** | **MODIFY** viene usata per modificare solo il tipo della colonna <NomeColonna> della tabella <NomeTabella>, non il suo nome.
- la clausola **CHANGE** viene usata per modificare il nome della colonna <NomeCol_OLD> della tabella <NomeTabella> nel nuovo nome <NomeCol_NEW> con eventualmente un nuovo tipo dati (obbligatorio).

Eliminare una tabella

Per cancellare completamente una tabella dalla base dei dati si utilizza l'istruzione **DROP TABLE** la cui sintassi è:

```
DROP TABLE <NomeTabella> [NO ACTION (analogo a RESTRICT) | CASCADE] ;
```

La cancellazione di una tabella può provocare inconsistenze dovute al fatto che possono esserci tabelle collegate tramite chiavi esterne o più in generale tramite vincoli di integrità.

Per far fronte a tali evenienze si utilizzano:

- l'istruzione **NO ACTION** che non permette la cancellazione se la tabella da cancellare è legata tramite vincoli di integrità ad altre tabelle (comportamento da non specificare perché di default);
- l'istruzione **CASCADE** che dà luogo ad una cancellazione ricorsiva in cascata di tutte le tabelle collegate;

N.B. queste opzioni al momento non sono state ancora sviluppate in MySQL

Creazione di un dominio

In SQL è possibile definire nuovi tipi di dato con l'istruzione **CREATE DOMAIN** la cui sintassi è:

```
CREATE DOMAIN <NomeDominio> AS <Tipo> [CHECK (<Condizione>) ] ;
```

dove

- si assegnano a <NomeDominio> tutti i valori di <Tipo> che verificano la <Condizione> (se presente perché opzionale).

Esistono, ovviamente, anche le istruzioni per:

Modificare un dominio

In SQL è possibile modificare un nuovo tipo di dato precedentemente creato, con l'istruzione **ALTER DOMAIN** la cui sintassi è:

```
ALTER DOMAIN <NomeDominio> AS <Tipo> [CHECK (<Condizione>) ] ;
```

Eliminare un dominio (DROP DOMAIN)

In SQL è possibile eliminare un nuovo tipo di dato precedentemente creato, con l'istruzione **DROP DOMAIN** la cui sintassi è:

```
DROP DOMAIN <NomeDominio> [NO ACTION (analogo a RESTRICT) | CASCADE] ;
```

Creazione di un indice

In SQL è possibile legare agli attributi di una tabella alcune **tabelle speciali dette indici**.

Tali indici sono **file** contenenti le chiavi delle tabelle alle quali sono associati che permettono al linguaggio SQL di accelerare il processo di ricerca dei dati all'interno della tabella ove sono immagazzinati.

Gli indici sono estranei al modello relazionale dei dati e sono molto vicini al **modello fisico dei dati**.

L'istruzione per creare un indice per una tabella è **CREATE INDEX** la cui sintassi è la seguente:

```
CREATE [UNIQUE] INDEX <NomeIndice>  
    ON <NomeTabella> (<NomeAttributo1>, <NomeAttributo2>, ... , <NomeAttributoN>) ;
```

dove

- se la clausola opzionale **UNIQUE E' SPECIFICATA**, viene creato un indice su attributi chiave;
- se la clausola opzionale **UNIQUE NON E' SPECIFICATA**, viene creato un indice su attributi NON chiave;

N.B. Occorre valutare bene il rapporto costi-benefici dell'utilizzo degli indici sulle tabelle valutando il tempo di risposta alle interrogazioni ed i ritardi durante le modifiche dei dati.

Eliminazione di un indice

Per eliminare un indice si utilizza l'istruzione **DROP INDEX** la cui sintassi è la seguente:

```
DROP INDEX <NomeIndice> ON <NomeTabella>;
```

Linguaggio SQL: Vincoli di integrità

Riassumendo quanto visto finora per creare e gestire i vincoli RELAZIONALI in SQL possiamo dire che:

- Per esprimere i **vincoli interni o intrarelazionali** (su singola ennupla o su più ennuple) utilizziamo durante la creazione di una tabella (istruzione **CREATE TABLE**):
 - la clausola **NOT NULL** per esprimere i vincoli sul valore di un attributo;
 - la clausola **PRIMARY KEY** per esprimere i vincoli sulle chiavi primarie;
 - la clausola **UNIQUE** per esprimere vincoli sulle chiavi candidate;
 - la clausola **CHECK** per esprimere vincoli di ennupla che coinvolgono uno o più attributi;
- Per esprimere i **vincoli interrelazionali o esterni referenziali** utilizziamo durante la creazione di una tabella (istruzione **CREATE TABLE**):
 - la clausola **FOREIGN KEY ... REFERENCES** per indicare le chiavi esterne con in aggiunta la clausola **NO ACTION (analogo a RESTRICT) | CASCADE | SET DEFAULT | SET NULL** per implementare una delle tre politiche:
 - *rifiuto* delle modifiche che violano un vincolo
 - *propagazione* a cascata delle modifiche
 - *impostazione al valore di default* delle chiavi delle tabelle interessate
 - *impostazione a NULL* delle chiavi delle tabelle interessate
- Per esprimere i **vincoli interrelazionali esterni non referenziali** non si utilizzano specificazioni di elementi dello schema inserendole all'interno di altre istruzioni SQL.

Essi vengono tradotti utilizzando apposite istruzioni SQL le **asserzioni** che sono esse stesse elementi dello schema.

Per creare un'asserzione si utilizza l'istruzione **CREATE ASSERTION** che ha la seguente sintassi:

```
CREATE ASSERTION <NomeAsserzione> CHECK (<Condizione>) ;
```

dove:

<Condizione> deve essere sempre verificata (ossia sempre vera) e qualsiasi modifica alle relazioni che la renda falsa deve essere rifiutata;

<NomeAsserzione> coincide per maggiore chiarezza con l'identificativo del vincolo interrelazionale o esterno NON referenziale cui si riferisce.

ISTRUZIONI DML del linguaggio SQL

Una volta creato lo schema relazionale tramite le istruzioni precedenti occorre poter inserire, modificare e cancellare i valori delle righe (ossia le ennuple) delle tabelle.

Inserire una riga in una tabella

Nel linguaggio SQL è possibile inserire i valori di una riga (ennupla) in una tabella utilizzando l'istruzione **INSERT** la cui sintassi è:

```
INSERT INTO <NomeTabella> [ (<NomeAttributo1> , <NomeAttributo2> ,..., <NomeAttributoN>) ]
VALUES (<Valore1> , <Valore 2> ,..., <Valore N>) ;
```

Se **NON E' presente** una lista degli attributi di <NomeTabella>, si intende che i valori specificati nella clausola **VALUES** verranno fatti corrispondere secondo ordine, tipo e numero **A TUTTI** quelli specificati al momento della creazione di <NomeTabella>.

Se **E' presente** una lista degli attributi di <NomeTabella>, si intende che i valori specificati nella clausola **VALUES** verranno fatti corrispondere secondo ordine, tipo e numero **SOLO A** quelli presenti nella lista fornita mentre gli attributi in essa non presenti verranno posti a NULL.

L'istruzione di **INSERT** permette, se necessario, di inserire i valori di più righe (ennuple) all'interno di una tabella utilizzando questa forma sintattica alternativa:

```
INSERT INTO <NomeTabella> [ (<NomeAttributo1> , <NomeAttributo2> ,..., <NomeAttributoN>) ]
VALUES (<Valore1> , <Valore 2> ,..., <Valore N>),
      (<Valore1> , <Valore 2> ,..., <Valore N>),
      .....
      (<Valore1> , <Valore 2> ,..., <Valore N>);
```

Modificare i valori delle righe in una tabella

Nel linguaggio SQL è possibile aggiornare una o più righe (ennuple) di una tabella utilizzando l'istruzione **UPDATE** la cui sintassi è:

```
UPDATE <NomeTabella>
SET   <NomeAttributo1> = <Espressione1>,
      <NomeAttributo2> = <Espressione2>,
      .....
      <NomeAttributoN> = <EspressioneN>
[ WHERE (<Condizione>) ] ;
```

dove verranno aggiornati **SOLO** gli attributi di <NomeTabella> presenti nella clausola **SET** con i valori delle corrispondenti espressioni fornite:

- se **E' presente** la clausola **WHERE**, in tutte le righe della tabella che la soddisfano;
- se **NON E' presente** la clausola **WHERE**, in tutte le righe della tabella.

Cancellare le righe di una tabella

Nel linguaggio SQL è possibile cancellare una o più righe (ennuple) di una tabella utilizzando l'istruzione **DELETE FROM** la cui sintassi è:

```
DELETE
FROM <NomeTabella>
[ WHERE (<Condizione>) ] ;
```

dove verranno eliminate

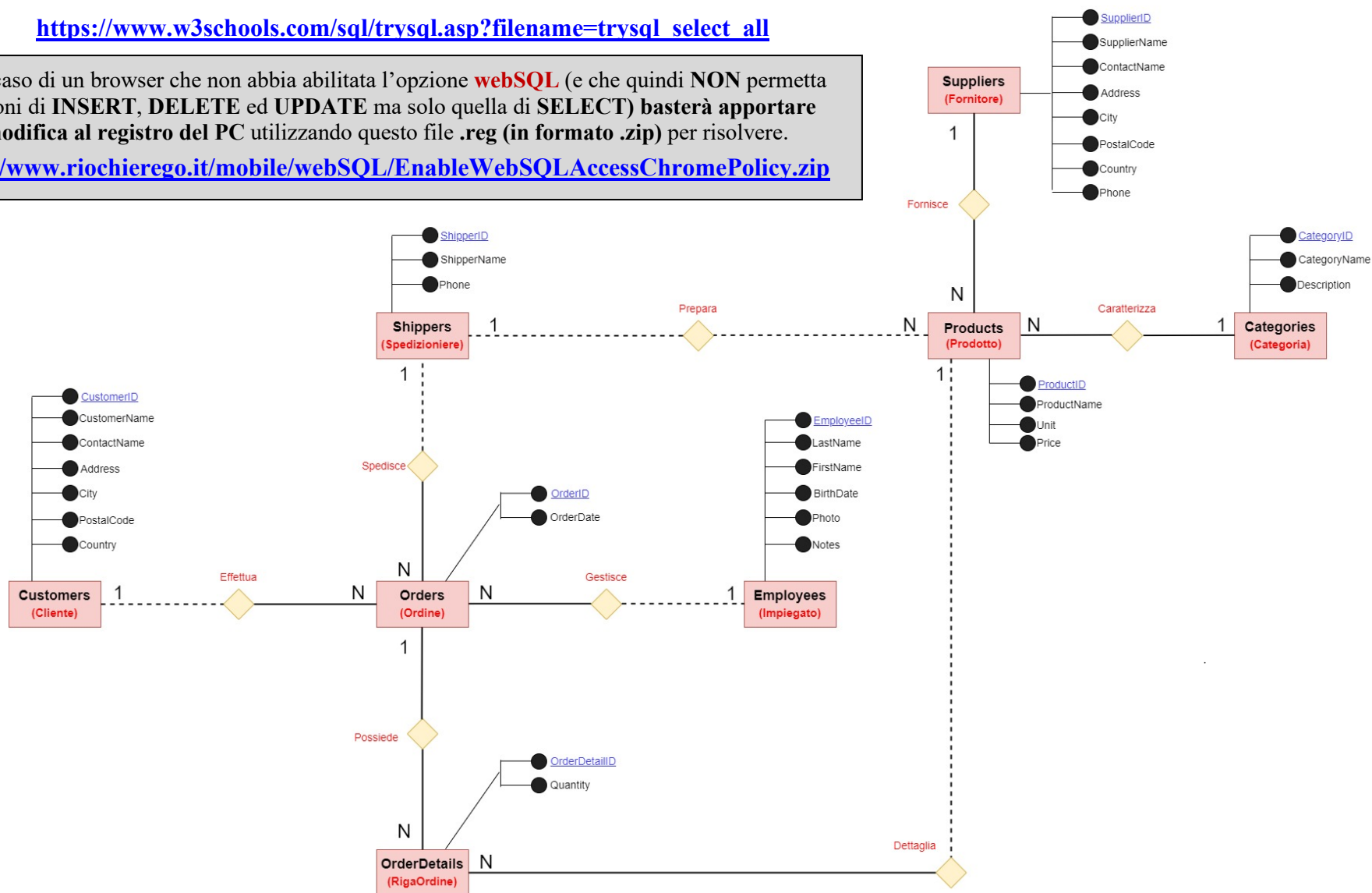
- se **E' presente** la clausola **WHERE**, solo le righe della tabella che la soddisfano;;
- se **NON E' presente** la clausola **WHERE**, tutte le righe della tabella.

N.B. Per potersi esercitare ON-LINE è possibile utilizzare ANCHE il database messo disposizione dall'organizzazione **W3schools** al seguente URL:

https://www.w3schools.com/sql/trysql.asp?filename=trysql_select_all

N.B. In caso di un browser che non abbia abilitata l'opzione **webSQL** (e che quindi **NON** permetta le istruzioni di **INSERT**, **DELETE** ed **UPDATE** ma solo quella di **SELECT**) basterà apportare questa modifica al registro del PC utilizzando questo file .reg (in formato .zip) per risolvere.

<https://www.riochierego.it/mobile/webSQL/EnableWebSQLAccessChromePolicy.zip>



ISTRUZIONI QL del linguaggio SQL

Reperire i dati attraverso una interrogazione o query: l'istruzione **SELECT**

Per reperire i dati il linguaggio SQL utilizza l'istruzione **SELECT** la cui potenza ed espressività è forse alla base del suo successo.

Il risultato di una qualunque interrogazione effettuata tramite la **SELECT** è a sua volta una *tabella* che viene mostrata a video oppure stampata, ma che può anche essere assegnata ad una variabile strutturata.

La sintassi dell'istruzione **SELECT** è **molto complessa**. Vediamo ora una sua **forma semplificata**:

```
SELECT [DISTINCT] <NomeAttributo1> [, <NomeAttributo2> , <NomeAttributoN> ]
FROM <NomeTabella1> [, <NomeTabella2>, ... , <NomeTabellaK> ]
[WHERE <Condizione> ] ;
```

L'istruzione **SELECT** restituisce una **tabella** formata dagli attributi

<NomeAttributo1> [, <NomeAttributo2> , <NomeAttributoN>]

della tabella (o del prodotto cartesiano delle tabelle)

<NomeTabella1> [, <NomeTabella2>, ... , <NomeTabellaK>]

ristretto alle righe che soddisfano l'eventuale <Condizione> espressa dalla clausola **WHERE** se presente, altrimenti se assente la <Condizione> si assume sempre vera.

Se è presente la clausola **DISTINCT** la tabella risultato sarà priva di righe duplicate.

Se si vogliono visualizzare tutti gli attributi presenti nel prodotto cartesiano delle tabelle è possibile indicare il simbolo '*' il cui significato sarà *"tutti gli attributi del prodotto delle tabelle"*.

La <Condizione> inoltre può essere un predicato composto nel senso dell'algebra di Boole (ossia composto attraverso l'uso dei connettivi logici **AND**, **NOT** ed **OR** da due o più predicati semplici)

Intestare le colonne della tabella risultato

Per *default* la tabella risultato di una **SELECT** ha come intestazione delle colonne il nome degli attributi della tabella.

Se si vuole dare un nome diverso ad ogni colonna del risultato si deve utilizzare la clausola **AS** chiamata **alias**.

Eseguire calcoli sulla tabella prodotta da una **SELECT** senza modificare il contenuto delle tabelle

E' possibile far eseguire all'istruzione **SELECT** il calcolo di una espressione sugli attributi mostrandola a video in una nuova colonna intestata con la clausola **AS**.

Il calcolo viene eseguito esternamente alla tabella senza modificare i dati in essa contenuti.

Esempio: Se vogliamo visualizzare una variazione del 10% degli stipendi dei dipendenti potremo scrivere

```
SELECT Cognome, Nome, Stipendio * 1,10 AS NuovoStipendio
FROM Dipendente;
```

Le operazioni relazionali in SQL attraverso l'utilizzo della SELECT

Le operazioni di *restrizione o selezione* (σ), *proiezione* (π) e *giunzione naturale* ($\bowtie$) su una base relazionale vengono realizzate attraverso l'istruzione **SELECT** grazie alle diverse forme consentite dalla sintassi di questa istruzione.

(A) L'operazione di restrizione o selezione (σ), che consente di ricavare da una relazione un'altra relazione contenente solo le righe che soddisfano una certa condizione, viene realizzata nel linguaggio SQL utilizzando l'istruzione **SELECT** considerando tutti i suoi attributi e specificando la clausola **WHERE** e

*Esempio: Se vogliamo l'elenco di tutti i dipendenti che hanno Stipendio maggiore o uguale a 1000,00 si opera una selezione o restrizione sulla tabella **Dipendente** ossia*

SELECT *
FROM Dipendente
WHERE Stipendio $\geq$ 1000.00 ;

che equivale, secondo la sintassi illustrata, alla seguente operazione dell'algebra relazionale

$$\sigma_{\text{StipendioNetto} \geq 1000,00}(\text{Dipendente}) \quad \xLeftrightarrow{\text{equivalentemente}} \quad \boxed{\begin{array}{l} \text{Se } R = \text{Dipendente} \quad e \quad P = \{ \text{Stipendio} \geq 1000.00 \} \\ \sigma_P(R) \end{array}}$$

(B) L'operazione di proiezione (π), che consente di ottenere una relazione contenente solo alcuni attributi della relazione di partenza, viene realizzata nel linguaggio SQL utilizzando l'istruzione **SELECT** impostando la clausola **DISTINCT** per eliminare eventuali ennuple duplicate e specificando l'elenco degli attributi richiesti ma senza impostare la clausola **WHERE** ed

*Esempio: Se vogliamo l'elenco di tutti i dipendenti visualizzando soltanto Cognome, Nome e Stipendio si opera una proiezione sulla tabella **Dipendente** ossia*

SELECT DISTINCT Cognome, Nome, Stipendio
FROM Dipendente;

che equivale, secondo la sintassi illustrata, alla seguente operazione dell'algebra relazionale

$$\pi_{\text{Cognome, Nome, Stipendio}}(\text{Dipendente}) \quad \xLeftrightarrow{\text{equivalentemente}} \quad \boxed{\begin{array}{l} \text{Se } R = \text{Dipendente} \quad e \quad A = \{ \text{Cognome, Nome, Stipendio} \} \\ \pi_A(R) \end{array}}$$

(C) L'operazione di prodotto cartesiano $\times$ su due (o più) tabelle viene realizzata nel linguaggio SQL utilizzando l'istruzione **SELECT** selezionando tutti gli attributi ma senza utilizzare la clausola **WHERE** ed indicando i nomi (separati da virgole) delle tabelle dopo la parola chiave **FROM**

(D) L'operazione di giunzione naturale o equi-join (chiamata anche **inner-join**) viene anch'essa realizzata nel linguaggio SQL utilizzando il comando **SELECT**.

In pratica si effettua prima il prodotto cartesiano come al punto (C) e poi una selezione come al punto (A) grazie alla clausola **WHERE** imponendo l'uguaglianza dei valori degli attributi che devono corrispondere nelle due tabelle.

Utilizzando gli **alias** (ossia la clausola **AS**) sarà possibile effettuare anche l'operazione di **SELF JOIN** (ossia join sulla stessa tabella)

(E) Le operazioni di **unione, intersezione e differenza**

Per tradurre le operazioni dell'algebra relazionale di *unione, intersezione e differenza* il linguaggio SQL utilizza tre operatori che si applicano ai risultati delle interrogazioni.

Tali operatori sono:

- **UNION** (per l'operazione relazionale di **unione** o $\cup$),
- **INTERSECT** (per l'operazione relazionale di **intersezione** o $\cap$)
- **MINUS** (per l'operazione relazionale di **differenza** o $-$)

(N.B. **INTERSECT** e **MINUS** non valide in **MySql**.)

Per poterle svolgere in modo equivalente, occorrerà utilizzare la tecnica delle sottointerrogazioni o sottoquery o SUBQUERY)

Esempio: Consideriamo le seguenti relazioni di un ipotetico db chiamato "cinema"

Regista (CodRegista, Cognome, Nome)

Attore (CodAttore, Cognome, Nome)

Per ottenere tutti i registi e tutti gli attori scriveremo

(**SELECT** Cognome, Nome **FROM** Regista)

UNION

(**SELECT** Cognome, Nome **FROM** Attore);

Per ottenere i registi che sono anche attori scriveremo

(**SELECT** Cognome, Nome **FROM** Regista)

INTERSECT

(**SELECT** Cognome, Nome **FROM** Attore);

Per ottenere i registi che **NON** sono attori scriveremo

(**SELECT** Cognome, Nome **FROM** Regista)

MINUS

(**SELECT** Cognome, Nome **FROM** Attore);

Naturalmente le due relazioni parziali delle **SELECT** dovranno **OBBLIGATORIAMENTE** risultare **compatibili** per poter essere utilizzate come operandi degli operatori di **INTERSEZIONE, UNIONE e DIFFERENZA**.

**Vedi paragrafo
sottointerrogazioni o sottoquery o
SUBQUERY
a partire da pagina 47**

(F) Le operazioni di **INNER JOIN, LEFT JOIN, RIGHT JOIN, OUTER JOIN**

N.B. E' possibile utilizzare nella **SELECT** direttamente le clausole **INNER JOIN, LEFT JOIN, RIGHT JOIN, OUTER JOIN** e **SELF JOIN**

(N.B. **OUTER JOIN** e **SELF JOIN** non valide in **MySql**)

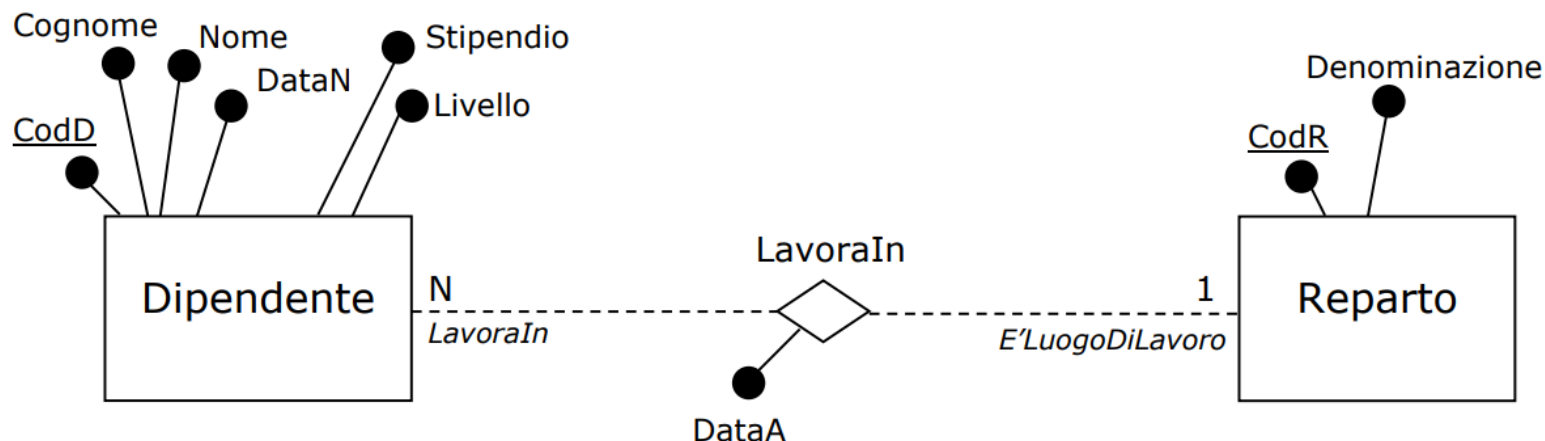
N.B. Vedi esempio di dettaglio a partire da pagina 17

ESEMPI di query tratte dall'ESERCITAZIONE: Dipendente-Reparto (N:1)

Per l'esempio completo si rimanda al link seguente:

<http://www.riochierego.it/mobile/docs/quinta/lab/ESERCITAZIONE-db-esempio-DIPENDENTE-REPARTO-SOLO-TESTO-QUERY-ALUNNO.pdf>

Diagramma ER



Mapping logico relazionale (in sintesi)

a) Mapping relazionale dell'associazione binaria "LavoraIn" tra le entità "Dipendente" e "Reparto" di molteplicità N:1

Dipendente (CodD, Cognome, Nome, DataN, Stipendio, Livello, DataA, CodR1)

con l'attributo "CodR1" della relazione "Dipendente" che risulta essere chiave esterna o Foreign Key (o FK) sull'attributo "CodR" della relazione "Reparto"

Reparto (CodR, Denominazione)

Supponiamo ora che le istanze presenti nelle sue due tabelle, **in un certo istante di tempo t**, siano le seguenti

Reparto	
CodR	Denominazione
R1	Fonderia
R2	Magazzino
R3	Amministrazione

Dipendente							
CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1
D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL

Eseguiamo alcune QUERY di esempio:

OPERATORE SELEZIONE o RESTRIZIONE σ

Esempio query: **Trovare i dipendenti nati dopo il 20-10-1968**

$P = \{ (DataN > "1968-10-20") \}$

σ_P (Dipendente)

SELECT *
FROM Dipendente
WHERE (DataN > "1968-10-20");

σ
SELEZIONE o RESTRIZIONE "PURA"

RISULTATO QUERY

σ_P (Dipendente)							
CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1
D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL

OPERATORE PROIEZIONE Π

Esempio query: **Trovare il cognome di tutti i dipendenti**

$A = \{ \text{Cognome} \}$
 $\Pi_A (\text{Dipendente})$

```
SELECT DISTINCT Cognome
FROM Dipendente;
```

Π
PROIEZIONE "PURA"

RISULTATO QUERY

$\Pi_A (\text{Dipendente})$
Cognome
Costanzo
Marini
Verstappen

N.B. se si desidera vederli comunque tutti SENZA ELIMINARE le ennuple DUPLICATE allora

```
SELECT ALL Cognome
FROM Dipendente;
```

Proiezione banale

Possibili duplicati

```
SELECT Cognome
FROM Dipendente;
```

**N.B. Queste non sono
PROIEZIONI**

Proiezione banale

```
SELECT *
FROM Dipendente;
```

OPERATORE PRODOTTO CARTESIANO X

Esempio query: **Eseguire il prodotto cartesiano delle relazioni Dipendente e Reparto**

Dipendente X Reparto

SELECT *
FROM Dipendente, Reparto

X
PRODOTTO CARTESIANO
"PURO"

RISULTATO QUERY

Dipendente X Reparto									
CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1	CodR	Denominazione
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1	R1	Fonderia
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1	R2	Magazzino
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1	R3	Amministrazione
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2	R1	Fonderia
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2	R2	Magazzino
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2	R3	Amministrazione
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1	R1	Fonderia
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1	R2	Magazzino
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1	R3	Amministrazione
D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL	R1	Fonderia
D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL	R2	Magazzino
D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL	R3	Amministrazione

Esempio query: **Eseguire il prodotto cartesiano delle relazioni Reparto e Dipendente**

Reparto X Dipendente

SELECT *
FROM Reparto, Dipendente;

X
PRODOTTO CARTESIANO
"PURO"

RISULTATO QUERY

Reparto X Dipendente									
CodR	Denominazione	CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1
R1	Fonderia	D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1
R1	Fonderia	D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2
R1	Fonderia	D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1
R1	Fonderia	D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL
R2	Magazzino	D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1
R2	Magazzino	D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2
R2	Magazzino	D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1
R2	Magazzino	D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL
R3	Amministrazione	D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1
R3	Amministrazione	D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2
R3	Amministrazione	D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1
R3	Amministrazione	D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL

N.B. L'OPERATORE RELAZIONALE DEL PRODOTTO CARTESIANO NON GODE DELLA PROPRIETA' COMMUTATIVA

OPERATORE EQUI-JOIN $\bowtie$ (NOTA BENE: il simbolo utilizzato si trova nel font di caratteri **Wingdings3**)

Esempio query: **Visualizzare i dati di tutti i dipendenti compresi i dati relativi ai reparti in cui lavorano**

Dipendente $\bowtie$ **Reparto**

$\text{CodR1} = \text{CodR}$

che, come è noto, risulta equivalente a

σ_P (**Dipendente** $\times$ **Reparto**)

con $P = \{ \text{CodR1} = \text{CodR} \}$

SELECT *

FROM Dipendente **INNER JOIN** Reparto **ON** $\text{CodR1} = \text{CodR}$;

oppure

SELECT *

FROM Dipendente, Reparto

WHERE ($\text{CodR1} = \text{CodR}$);

$\bowtie$
EQUI-JOIN "PURO"

RISULTATO QUERY

Dipendente $\bowtie$ Reparto $\text{CodR1} = \text{CodR}$									
CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1	CodR	Denominazione
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1	R1	Fonderia
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2	R2	Magazzino
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1	R1	Fonderia

Esempio query: **Visualizzare i dati di tutti i reparti compresi i dati relativi ai dipendenti che ci lavorano**

Reparto $\bowtie$ Dipendente

CodR = CodR1

che, come è noto, risulta equivalente a

σ_P (Reparto $\times$ Dipendente)

con $P = \{ \text{CodR} = \text{CodR1} \}$

SELECT *
FROM Reparto **INNER JOIN** Dipendente **ON** CodR = CodR1;
 oppure

SELECT *
FROM Reparto, Dipendente
WHERE (CodR = CodR1);

$\bowtie$
EQUI-JOIN "PURO"

RISULTATO QUERY

Reparto $\bowtie$ Dipendente CodR = CodR1									
CodR	Denominazione	CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1
R1	Fonderia	D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1
R1	Fonderia	D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1
R2	Magazzino	D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2

N.B. L'OPERATORE RELAZIONALE DI EQUI-JOIN NON GODE DELLA PROPRIETA' COMMUTATIVA

(derivando ovviamente in parte dal prodotto cartesiano)

Esempio query: Visualizzare i dati di tutti i dipendenti con i dati relativi ai reparti in cui **NON** lavorano

Dipendente $\bowtie$ Reparto

CodR1 $\bowtie$ CodR

che, come è noto, risulta equivalente a

σ_P (Dipendente $\times$ Reparto)

con $P = \{ \text{CodR1} \bowtie \text{CodR} \}$

SELECT *

FROM Dipendente **INNER JOIN** Reparto **ON** CodR1 $\bowtie$ CodR;

oppure

SELECT *

FROM Dipendente, Reparto

WHERE (CodR1 $\bowtie$ CodR);

$\bowtie$

**TETHA-JOIN
"PURO"**

RISULTATO QUERY

Dipendente $\bowtie$ Reparto CodR1 $\bowtie$ CodR									
CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1	CodR	Denominazione
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1	R2	Magazzino
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1	R3	Amministrazione
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2	R1	Fonderia
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2	R3	Amministrazione
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1	R2	Magazzino
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1	R3	Amministrazione

N.B. Mancano le enuple con CodR1 = NULL in quanto il confronto in questo caso ha esito NEGATIVO

OPERATORE LEFT-OUTER-JOIN ^{SX} ▷◁ (NOTA BENE: carattere Wingdings3)

Esempio query: **Visualizzare i dati di tutti i dipendenti con i dati relativi ai reparti in cui lavorano**, compresi ANCHE i dati dei dipendenti che non lavorano ancora in alcun reparto

^{SX}
Dipendente ▷◁ Reparto
CodR1 = CodR

SELECT *
FROM Dipendente **LEFT JOIN** Reparto **ON** CodR1 = CodR;

▷◁
LEFT-OUTER-JOIN
"PURO"

RISULTATO QUERY

^{SX} Dipendente ▷◁ Reparto CodR1 = CodR									
CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1	CodR	Denominazione
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1	R1	Fonderia
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2	R2	Magazzino
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1	R1	Fonderia
D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL	NULL	NULL

OPERATORE RIGHT-OUTER-JOIN ^{DX}
 $\triangleright\triangleleft$ (NOTA BENE: carattere Wingdings3)

Esempio query: **Visualizzare i dati di tutti i dipendenti con i dati relativi ai reparti in cui lavorano,** compresi ANCHE i dati dei reparti che in cui non lavora alcun dipendente

^{DX}
Dipendente $\triangleright\triangleleft$ **Reparto**
 CodR1 = CodR

SELECT *
FROM Dipendente **RIGHT JOIN** Reparto **ON** CodR1 = CodR;

$\triangleright\triangleleft$
RIGHT-OUTER-JOIN
"PURO"

RISULTATO QUERY

^{DX} Dipendente $\triangleright\triangleleft$ Reparto CodR1 = CodR									
CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1	CodR	Denominazione
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1	R1	Fonderia
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2	R2	Magazzino
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1	R1	Fonderia
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	R3	Amministrazione

OPERATORE FULL-OUTER-JOIN ^E (NOTA BENE: carattere Wingdings3)

Esempio query: **Visualizzare i dati di tutti i dipendenti con i dati relativi ai reparti in cui lavorano**, compresi ANCHE i dati dei dipendenti che non lavorano ancora in alcun reparto e compresi ANCHE i dati dei reparti che in cui non lavora alcun dipendente

N.B. Non essendoci l'operatore OUTER JOIN nel db server MySql che utilizziamo possiamo ottenere le ennuple richieste eseguendo un'operazione alternativa (l'UNIONE tra le tabelle ottenute dopo un LEFT JOIN ed un RIGHT JOIN)

^E
Dipendente  **Reparto**
 CodR1 = CodR

SELECT *
 FROM Dipendente **OUTER JOIN** Reparto **ON** Dipendente.CodR1 = Reparto.CodR)

(SELECT *
 FROM Dipendente **LEFT JOIN** Reparto **ON** CodR1 = CodR)
UNION
 (SELECT *
 FROM Dipendente **RIGHT JOIN** Reparto **ON** CodR1 = CodR);

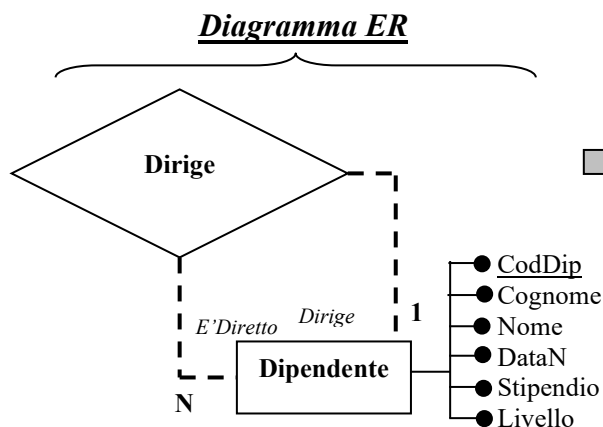

FULL-OUTER-JOIN
"PURO"

RISULTATO QUERY

^E Dipendente  Reparto CodR1 = CodR									
CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1	CodR	Denominazione
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1	R1	Fonderia
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2	R2	Magazzino
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1	R1	Fonderia
D4	Verstappen	Max	1980-11-12	4200.00	9	NULL	NULL	NULL	NULL
NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	R3	Amministrazione

ESERCITAZIONE: Associazione binaria RICORSIVA "Dirige" sulla stessa entità "Dipendente" di molteplicità 1:N

Supponiamo di avere il seguente diagramma ER (con il relativo mapping relazionale) dell'associazione binaria ricorsiva "Dirige" di molteplicità 1:N sulla stessa tabella Dipendente



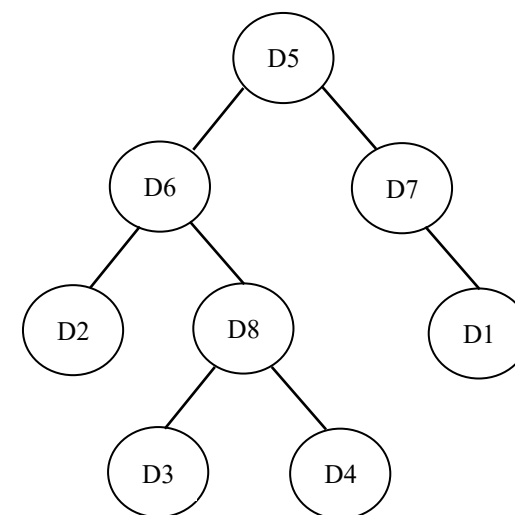
Dipendente(CodDip , Cognome, Nome, DataN, Stipendio, Livello, CodCapo)

- l'attributo "CodCapo" della relazione "Dipendente" è CHIAVE ESTERNA sull'attributo "CodDip" della stessa relazione "Dipendente"

Mapping logico relazionale (in sintesi)

Supponiamo ora di avere creato il database "Db_Dipendente_Ric" e che le istanze presenti nella sua unica tabella, **in un certo istante di tempo t**, siano le seguenti

Dipendente						
CodDip	Cognome	Nome	DataN	Stipendio	Livello	CodCapo
D1	Costanzo	Renato	1950-08-25	1200.00	7	D7
D2	Marini	Valeria	1965-11-09	1350.00	7	D6
D3	Costanzo	Maurizio	1975-03-13	2350.00	6	D8
D4	Verstappen	Max	1980-11-12	1280.00	6	D8
D5	Chierego	Rio	1965-09-08	4750.00	9	NULL
D6	Altruda	Giustino	1960-12-13	1650.00	8	D5
D7	Alicante	Anita	1975-05-24	1100.00	8	D5
D8	Mosca	Anna	1964-06-08	2200.00	7	D6

GERARCHIA AZIENDALE

OPERATORE EQUI-JOIN $\bowtie$ (NOTA BENE: carattere Wingdings3)

Esempio query: **Visualizzare i dati di tutti i dipendenti con i dati relativi ai loro rispettivi capi**

Dipendente $\bowtie$ **Dipendente**

CodCapo = CodDip

che, come è noto, risulta equivalente a

 σ_P (Dipendente X Dipendente)con $P = \{ \text{CodCapo} = \text{CodDip} \}$ **SELECT *****FROM** Dipendente **AS** Tab1, Dipendente **AS** Tab2**WHERE** (Tab1.CodCapo = Tab2.CodDip);
 $\bowtie$
SELF-JOIN
"PURO"
OPPURE**SELECT** Tab1.Cognome, Tab1.Nome, Tab2.Cognome, Tab2.Nome**FROM** Dipendente **AS** Tab1 **INNER JOIN** Dipendente **AS** Tab2 **ON** Tab1.CodCapo = Tab2.CodDip;**RISULTATO QUERY**

CodDip	Cognome	Nome	DataN	Stipendio	Livello	CodCapo	CodDip	Cognome	Nome	DataN	Stipendio	Livello	CodCapo
D1	Costanzo	Renato	1950-08-25	1200.00	7	D7	D7	Alicante	Anita	1975-05-24	1100.00	8	D5
D2	Marini	Valeria	1965-11-09	1350.00	7	D6	D6	Altruda	Giustino	1960-12-13	1650.00	8	D5
D3	Costanzo	Maurizio	1975-03-13	2350.00	6	D8	D8	Mosca	Anna	1964-06-08	2200.00	7	D6
D4	Verstappen	Max	1980-11-12	4200.00	6	D8	D8	Mosca	Anna	1964-06-08	2200.00	7	D6
D6	Altruda	Giustino	1960-12-13	1650.00	8	D5	D5	Chierego	Rio	1965-09-08	1750.00	9	NULL
D7	Alicante	Anita	1975-05-24	1100.00	8	D5	D5	Chierego	Rio	1965-09-08	1750.00	9	NULL
D8	Mosca	Anna	1964-06-08	2200.00	7	D6	D6	Altruda	Giustino	1960-12-13	1650.00	8	D5

OPERATORE LEFT-OUTER-JOIN  SX (NOTA BENE: carattere Wingdings3)

Esempio query **Visualizzare i dati di tutti i dipendenti con i dati relativi ai loro rispettivi capi, ivi compreso quelli relativi al capo assoluto (il capo dei capi... 😊)**

SX
Dipendente  **Dipendente**
CodCapo = CodDip

SELECT *
FROM Dipendente AS Tab1 **LEFT JOIN** Dipendente AS Tab2
ON (Tab1.CodCapo = Tab2.CodDip);

 **LEFT-OUTER-JOIN**
"PURO"

RISULTATO QUERY

CodDip	Cognome	Nome	DataN	Stipendio	Livello	CodCapo	CodDip	Cognome	Nome	DataN	Stipendio	Livello	CodCapo
D1	Costanzo	Renato	1950-08-25	1200.00	7	D7	D7	Alicante	Anita	1975-05-24	1100.00	8	D5
D2	Marini	Valeria	1965-11-09	1350.00	7	D6	D6	Altruda	Giustino	1960-12-13	1650.00	8	D5
D3	Costanzo	Maurizio	1975-03-13	2350.00	6	D8	D8	Mosca	Anna	1964-06-08	2200.00	7	D6
D4	Verstappen	Max	1980-11-12	4200.00	6	D8	D8	Mosca	Anna	1964-06-08	2200.00	7	D6
D5	Chierego	Rio	1965-09-08	1750.00	9	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL
D6	Altruda	Giustino	1960-12-13	1650.00	8	D5	D5	Chierego	Rio	1965-09-08	1750.00	9	NULL
D7	Alicante	Anita	1975-05-24	1100.00	8	D5	D5	Chierego	Rio	1965-09-08	1750.00	9	NULL
D8	Mosca	Anna	1964-06-08	2200.00	7	D6	D6	Altruda	Giustino	1960-12-13	1650.00	8	D5

Esempio query **Visualizzare i dati ESCLUSIVAMENTE del capo assoluto**

σ_P (Dipendente)
con $P = \{ \text{CodCapo IS NULL} \}$

SELECT *
FROM Dipendente
WHERE CodCapo IS NULL;

oppure, come vedremo più avanti, con una SUBQUERY

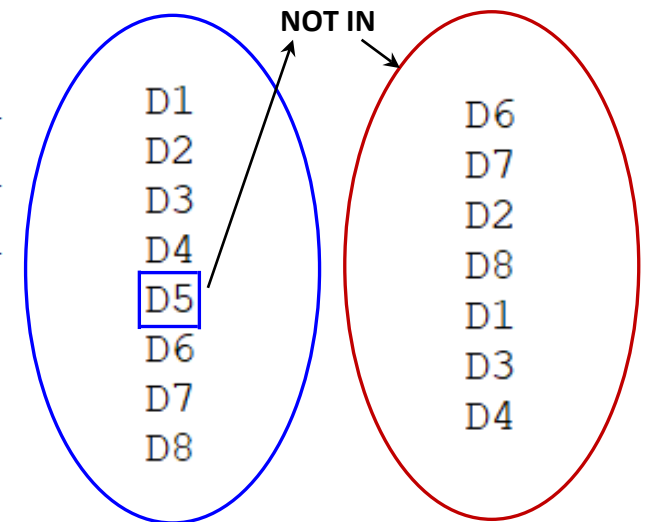
SELECT *
FROM Dipendente
WHERE CodDip NOT IN (**SELECT** Tab1.CodDip
FROM Dipendente AS Tab1, Dipendente AS Tab2
WHERE (Tab1.CodCapo = Tab2.CodDip));

Query

Subquery

RISULTATO QUERY

CodDip	Cognome	Nome	DataN	Stipendio	Livello	CodCapo
D5	Chierago	Rio	1965-09-08	1750.00	9	NULL



Esempio query: Visualizzare il cognome ed il nome dei dipendenti con accanto il cognome ed il nome dei relativi "capi"

$\Pi_A (\text{Dipendente} \triangleright \triangleleft \text{Dipendente})$

CodCapo = CodDip

che, come è noto, risulta equivalente a

$\Pi_A (\sigma_P (\text{Dipendente} \times \text{Dipendente}))$

con $P = \{ \text{CodCapo} = \text{CodDip} \}$

con $A = \{ \text{Tab1.Cognome}, \text{Tab1.Nome}, \text{Tab2.Cognome}, \text{Tab2.Nome} \}$

SELECT Tab1.Cognome, Tab1.Nome, Tab2.Cognome, Tab2.Nome
FROM Dipendente **AS** Tab1, Dipendente **AS** Tab2
WHERE (Tab1.CodCapo = Tab2.CodDip);

RISULTATO QUERY

Cognome	Nome	Cognome	Nome
Costanzo	Renato	Alicante	Anita
Marini	Valeria	Altruda	Giustino
Costanzo	Maurizio	Mosca	Anna
Verstappen	Max	Mosca	Anna
Altruda	Giustino	Chierego	Rio
Alicante	Anita	Chierego	Rio
Mosca	Anna	Altruda	Giustino

Le interrogazioni parametriche

Pur **non supportate dallo standard ANSI** le interrogazioni parametriche sono molto utili quando si utilizza l'SQL in modalità stand-alone.

E' possibile parametrizzare una query in modo da scriverla una volta sola e sfruttarla per diversi valori del parametro. Per far ciò basta far precedere la query dalla seguente dichiarazione:

```
PARAMETERS    <Parametro1>[:<TipoParametro1>],  
               <Parametro2>[:<TipoParametro2>],  
               .....  
               <ParametroN>[:<TipoParametroN>];
```

Esempio

PARAMETERS **Nome**, **Data**:DATE

SELECT *

FROM Cliente

WHERE (NomeCliente = **Nome**) AND (DataNascita = **Data**);

Verrà richiesto di inserire:

- il valore del nome del cliente sul quale il sistema non effettuerà alcun controllo;
- la data di nascita del cliente sulla quale il sistema effettuerà un controllo di tipo (avendolo esplicitamente utilizzato nell'istruzione **PARAMETERS**)

In ambiente Microsoft (Access) una query parametrica può essere del tipo:

SELECT *

FROM Dipendente

WHERE Cognome = [**ValoreX**];

In ambiente MySQL la sintassi è diversa da quella utilizzata in ambiente Microsoft (Access).

E' possibile eseguire query parametriche in MySQL ma **non possono essere usate** le parentesi []. Bisogna passare attraverso il comando SET che consente l'uso di variabili.

Esempio:

Per trovare tutti i cognomi il cui contenuto è uguale alla variabile @ValoreX sono necessari i comandi SQL:

SET **@ValoreX**='COSTANZO';

SELECT *

FROM Dipendente

WHERE Cognome = **@ValoreX**;

alla variabile **ValoreX** viene assegnato il valore 'COSTANZO'.

Nota Bene E' necessario l' uso di @ per assegnare il valore.

Eseguire QUERY CASE SENSITIVE in SQL (tabella case sensitive)

PREMESSA

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

Ogni database relazionale come MySQL o MariaDB utilizza i **CHARACTER SET** per gestire come i caratteri vengono codificati e salvati nei campi delle tabelle.

In ogni **CHARACTER SET**, ad un valore binario corrisponde un carattere ben preciso.

Di conseguenza, quando una stringa viene memorizzata utilizzando un certo insieme di caratteri, **dovrà essere visualizzata attraverso quello stesso insieme**, altrimenti alcuni caratteri potrebbero apparire diversi da come ce li aspettiamo.

L'esempio classico di questo inconveniente si verifica in genere con le lettere accentate e, a volte, con il simbolo dell'Euro, che ogni tanto capita di vedere non rappresentati correttamente, ad esempio su una pagina web: segno che non stiamo visualizzando quella pagina con il giusto insieme di caratteri.

MySQL, a partire dalla versione 4.1, ha introdotto un supporto molto avanzato alla gestione di diversi **CHARACTER SET**.

Infatti ci consente di gestire i set di caratteri a livello di **server**, **database**, **tabella** e **singola colonna**, nonché di **client** e di **connessione**.

Una **COLLATION** è un insieme di regole che determinano come confrontare e ordinare i caratteri in un database. Ogni **CHARACTER SET** ha una o più **COLLATION** associate, che specificano come i caratteri vengono confrontati per operazioni come **ORDER BY**, **GROUP BY** o per eseguire confronti di uguaglianza.

Quando in MySQL si crea un qualsiasi database, **PER DEFAULT** esso utilizzerà il **CHARACTER SET latin1** (quello maggiormente usato in Europa Occidentale) e la **COLLATION latin1_swedish_ci** per tutte le tabelle in esso contenute (se non diversamente specificato)

Tale **COLLATION** è **multilingue** (cioè non specifica per una lingua) e **"case insensitive"** come dimostra il **"ci"** finale, cioè tiene conto della differenza fra maiuscole e minuscole nell'ordinare o confrontare le stringhe.

In generale possiamo dire che il nome di ogni **COLLATION** segue un determinato standard:

- inizia con il **nome** del **CHARACTER SET** a cui si riferisce,
- comprende di solito una specifica relativa ad una lingua,
- termina con **cs** (**case sensitive**) o **ci** (**case insensitive**) a seconda che tenga o meno in considerazione la differenza tra maiuscole e minuscole, oppure con **bin** quando il valore binario dei caratteri è utilizzato direttamente per i confronti.

Normalmente l'esistenza di una collation dedicata ad una singola lingua si ha quando le regole generali del set di caratteri non soddisfano le esigenze di quella lingua. Le collation dedicate al tedesco, ad esempio, servono a trattare nel giusto modo le vocali con dieresi (ad esempio Ü) e la lettera "sharp" (ß)

Le istruzioni SQL **"SHOW CHARACTER SET;"** e **"SHOW COLLATION;"** ci consentono di ottenere la lista, rispettivamente, dei **CHARACTER SET** e delle **COLLATION** disponibili sul server.

In mancanza di inizializzazione, come già detto, il CHARACTER SET di default e la relativa COLLATION dipenderanno dalle impostazioni del server (in genere CHARACTER SET latin1 e COLLATION di default latin1_swedish_ci).

N.B. Se volessimo conoscere il CHARACTER SET di una certa tabella possiamo utilizzare l'istruzione:

```
SHOW [FULL] COLUMNS FROM nome_tabella [FROM nome_db]

[LIKE 'pattern' | WHERE espr]
```

SHOW COLUMNS è un'istruzione in grado di mostrare le informazioni sulle colonne di una data tabella e funziona anche con le viste.

La clausola **LIKE**, se presente, restringe l'output alle colonne i cui nomi corrispondono al pattern. La clausola **WHERE** può essere usata per specificare condizioni più generiche, come spiegato in <http://dev.mysql.com/doc/refman/5.1/en/extended-show.html>.

Immaginiamo di volere effettuare delle **QUERY case-sensitive** in MySql su tabelle appositamente create a tale scopo ottenute con **COLLATE case-sensitive**

Esempio:

Per quanto detto prima prendiamo in considerazione un **database case sensitive** costruito con la seguente istruzione eseguita dalla shell di MySQL:

```
CREATE DATABASE db_cs CHARACTER SET latin1 COLLATE latin1_general_cs;
```

contenente l'unica tabella Cliente creata con la seguente istruzione eseguita anch'essa dalla shell di MySQL:

```
CREATE TABLE Cliente
```

```
(
  CodC      VARCHAR(10) NOT NULL,
  Cognome   VARCHAR(30) NOT NULL,
  Nome      VARCHAR(30) NOT NULL,
  PRIMARY KEY (CodC)
) Engine InnoDB CHARACTER SET latin1 COLLATE latin1_general_cs;
```

Ipotizziamo che in questa tabella siano presenti **due clienti**, uno con cognome "**Rossi**" e l'altro con tutte le lettere minuscole "**rossi**".

Il cognome è lo stesso ma il dato è differente.

Se eseguiamo la seguente query:

```
SELECT *
FROM Cliente
WHERE Cognome = "Rossi";
```

quest'ultima visualizzerà soltanto i dati del cliente con cognome uguale a "**Rossi**" escludendo l'altro.

Per consentire la visualizzazione dei dati di entrambi i record ("**Rossi**" e "**rossi**") si devono operare delle accortezze all'interno della **SELECT** utilizzando le funzioni SQL **LOWER** e **UPPER**.

Le funzioni **LOWER** e **UPPER** trasformano rispettivamente il dato in lettere minuscole o maiuscole.

Quindi entrambe le query:

```
SELECT *
FROM Cliente
WHERE LOWER(Cognome) = "rossi";
```

oppure

```
SELECT *
FROM Cliente
WHERE UPPER(Cognome) = "ROSSI";
```

permetteranno la visualizzazione dei dati di entrambi i clienti con cognome uguale a "**Rossi**" oppure a "**rossi**" senza l'esclusione di nessuno dei due.

Immaginiamo di volere effettuare delle **QUERY case-sensitive** in MySql **a prescindere dalla natura delle tabelle** in esso contenute, quindi anche in caso di **COLLATION case-insensitive**

Le query che prevedono le **condizioni WHERE o LIKE** senza gli opportuni accorgimenti eseguono un **confronto case insensitive**, ovvero non fanno alcuna distinzione tra **maiuscole e minuscole**.

Ribadiamo che ciò avviene in base alle **impostazioni previste di default da MySql** quando si crea una nuova tabella riguardanti **CHARACTER SET** e **COLLATE** che vengono impostati in modo **da non differenziare** maiuscole e minuscole.

Eppure, in alcuni ambiti le lettere maiuscole e minuscole possono essere molto importanti: pensiamo, ad esempio, al caso in cui desidera tenerne conto di ciò nella procedura di **verifica password** in un sistema di login (a cui però non è stato applicato un hash come md5 o sha1).

Inoltre, in tali casi il confronto **equipara anche le lettere con accenti diversi** tale per cui 'e', 'è' ed 'é' sono indistinte.

La soluzione a tale problematica da un punto di vista pratico è semplicissima.

Basterà usare all'interno delle nostre query **la funzione BINARY**.

La funzione BINARY

Questa funzione consente di eseguire un **confronto fra stringhe non in base ai caratteri ma byte per byte**, ovvero un confronto della codifica binaria.

Occorre ulteriormente precisare che tale funzione **tiene conto anche di eventuali spazi vuoti** e quindi una stringa 'a' sarà diversa da 'a '.

```
SELECT *  
FROM <Tabella>  
WHERE BINARY (<Nome_Colonna>) = '<valore_Colonna>' ;
```

N.B. Le parentesi tonde alla funzione BINARY posso essere omesse

IN CONCLUSIONE

Tutto ciò solitamente avviene in quanto il confronto riguarda **campi in formato testuale** (CHAR, VARCHAR e TEXT) in cui si è scelto l'impostazione di default per lo **CHARACTER SET** e la **COLLATION** che sono rispettivamente **latin1** e **latin1_swedish_ci** (comunque sarebbe meglio utilizzare la collation **utf8**)

Quindi, quando ci si trova in fase di creazione di una tabella, occorrerebbe tener conto anche di come si desidera poi effettuare i confronti.

Nella scelta fra i diversi tipi di **COLLATION** occorre **porre attenzione ai suffissi quali ci, cs e bin** i quali stanno appunto a significare rispettivamente **case insensitive**, **case sensitive** ed **binary**.

Esistono, in pratica, set di caratteri più "*precisi*" rispetto ad altri che invece equiparano tra loro maiuscole, minuscole ed accentate senza fare alcuna differenza

Bisogna fare molta attenzione in quanto scegliere **uno CHARACTER SET e COLLATION più preciso** determina un **maggiore spazio occupato nel database** e, di conseguenza, **query che richiedono confronti più lente**.

Il valore NULL

Il linguaggio SQL prevede un particolare simbolo indicato con **NULL** che viene utilizzato per indicare diverse situazioni (ad esempio quando il valore esiste ma è sconosciuto oppure quando il valore non esiste).

Il valore del simbolo **NULL** assume un ruolo significativo nel risultato di una espressione logica o aritmetica.

Esistono alcune **regole fondamentali** da ricordare:

- un'**espressione aritmetica** ha come risultato un valore sconosciuto (**UNKNOWN**) se un operando ha valore **NULL**;
- il **confronto** tra un valore **NULL** ed un qualsiasi altro valore (**NULL** compreso) produce sempre un valore **UNKNOWN**;
- il valore **NULL** non è una costante (quindi non può apparire in una espressione);
- se il predicato della clausola **WHERE** ha valore **UNKNOWN** la ennupla non viene considerata;
- nelle funzioni aggregate in generale le righe con valore **NULL** dell'attributo considerato non vengono considerate;
- il valore **UNKNOWN** è un valore di verità come **TRUE** e **FALSE**

Vediamo ora i valori di verità degli operatori **AND**, **OR**, **NOT** in una **logica a tre valori** (**TRUE**, **FALSE**, **UNKNOWN**):

AND	FALSE	TRUE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
TRUE	FALSE	TRUE	UNKNOWN
UNKNOWN	FALSE	UNKNOWN	UNKNOWN

OR	FALSE	TRUE	UNKNOWN
FALSE	FALSE	TRUE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
UNKNOWN	UNKNOWN	TRUE	UNKNOWN

VALORI	NOT
FALSE	TRUE
TRUE	FALSE
UNKNOWN	FALSE

Attenzione al significato di **NULL** in quanto l'espressione

WHERE Stipendio = NULL

è errata nella sintassi non essendo il valore **NULL** una costante, mentre l'espressione

WHERE Stipendio IS NULL è sintatticamente corretta.

N.B. **NULL** in pratica non è considerato un valore ma un simbolo che indica "*valore mancante*".

Nelle query per controllare se il valore di un attributo è presente oppure mancante (ossia è uguale a **NULL**) si ricorre ai predicati **IS NULL** ed **IS NOT NULL** nelle condizioni della clausola **WHERE**.

Gli operatori **LIKE** e **NOT LIKE**

Un operatore piuttosto "particolare" è **LIKE** il quale consente di effettuare dei "paragoni di somiglianza", che permette di restituire tutti i record/righe che *matchano* (corrispondono) secondo un modello di corrispondenza (*chiamiamolo **stringa_ricerca***)

L'operatore **LIKE** utilizza, per costruire tale modello di corrispondenza, due caratteri speciali da inserire dentro la **stringa_ricerca**:

% (percentuale): questo simbolo sostituisce un insieme di caratteri in una stringa.

Esempio. Con l'operatore di confronto LIKE la stringa di ricerca "Ro%" seleziona tutte le parole di qualsiasi lunghezza che iniziano per "Ro" e finiscono in "a". Ad esempio, Roma, Romagna, Romania, ecc.

_ (underscore): questo simbolo sostituisce un unico carattere in una stringa.

Esempio. Con l'operatore di confronto LIKE la stringa di ricerca "Ro_" seleziona le parole di 4 lettere che iniziano per "Ro" e terminano in "a". Ad esempio, Roma, Rosa, ecc.

Sintassi

```
SELECT <Attributo_1> [, <Attributo_2>, ... <Attributo_N>]
FROM <Tabella_1> [, <Tabella_2>, ... <Tabella_N>]
WHERE (Attributo LIKE stringa_ricerca) [ESCAPE '<carattere_escape>'];
```

N.B. I caratteri speciali **%** e **_** devono essere inseriti dentro la stringa di ricerca, all'interno degli apici singoli o dei doppi apici

La specifica **ESCAPE '<carattere_escape>'** si utilizza se può capitare il caso in cui il carattere underscore o la percentuale sia compreso nei dati della tabella.

In MySQL è possibile anche anteporre al carattere **%** o **_** il backslash ****.

L'operatore **NOT LIKE** è l'esatta negazione di ciò che restituisce l'operatore **LIKE**, cioè restituirà tutti i record / righe che non corrispondono secondo il modello di corrispondenza *stringa ricerca*

Supponiamo ad esempio di voler estrarre dalla nostra tabella "Amico" tutti i record corrispondenti ad un nome che inizia per "Maria":

```
SELECT *
FROM Amico
WHERE (Nome LIKE 'Maria%');
```

Questa query restituirà nomi del tipo:

```
Maria
Maria Rosa
Maria Antonietta
```

Se invece avessimo scritto la clausola **WHERE** in questo modo:

```
SELECT *
FROM Amico
WHERE (Nome LIKE '%Maria');
```

Avremmo ottenuto in risposta:

```
Maria
Anna Maria
```

La differenza, come si può notare, è data dalla posizione del carattere percentuale (%) che sta ad indicare rispettivamente "qualsiasi carattere dopo" e "qualsiasi carattere prima".

Se invece avessimo scritto la clausola **WHERE** in questo modo:

```
SELECT *  
FROM Amico  
WHERE (Nome LIKE '%Maria%');
```

avremmo ottenuto in risposta tutti i nomi che contengono "Maria" all'inizio, in mezzo o alla fine della stringa

```
Maria Rosa  
Anna Maria  
Anna Maria Rosa
```

Se ipotizziamo che alcuni nomi siano scritti nel seguente modo "Anna_Laura", "Anna_Maria" in questi casi è necessario utilizzare nell'interrogazione la specifica **ESCAPE**, introdotta dall'ANSI per fare in modo che l'SQL interpreti il carattere '_' sia come escape sia come carattere effettivo e non come carattere jolly.

```
SELECT *  
FROM Amico  
WHERE (Nome LIKE 'Anna!_M%') ESCAPE '!';
```

oppure

```
SELECT *  
FROM Amico  
WHERE (Nome LIKE 'Anna\ _M%');
```

LE FUNZIONI o OPERATORI DI AGGREGAZIONE

Il linguaggio SQL possiede alcune **funzioni o operatori di aggregazione** predefinite utilissime in molte circostanze in cui occorre effettuare conteggi, somme, calcoli, medie o altro ancora.

L'aggregazione è una forma di interrogazione attraverso cui si ottengono risultati riepilogativi del contenuto di una tabella; a tale scopo si utilizzano delle funzioni speciali che restituiscono un solo valore, e come tali concorrono a creare un'unica riga.

Tali funzioni o operatori si applicano ad una colonna di una tabella.

In SQL è possibile procedere all'aggregazione delle righe di una tabella, utilizzando all'interno dell'istruzione **SELECT** le seguenti funzioni o operatori:

1) **Funzione COUNT**

SINTASSI:

COUNT([DISTINCT] <NomeAttributo>) [AS <Alias>]

N.B. Nell'utilizzo nell'ambiente **MySQL** è **sintatticamente errato** lasciare uno spazio tra il nome della funzione di aggregazione e la prima parentesi tonda aperta!

La funzione **COUNT** può essere usata su tutti i tipi di dati e conta il numero di elementi (righe) presenti in una tabella in un certo istante di tempo ossia restituisce la **cardinalità** di una relazione.

La sintassi del linguaggio SQL richiede di specificare come argomento della funzione il nome di un attributo oppure il carattere * (**asterisco**):

- nel primo caso non vengono conteggiate le righe che hanno valore **NULL** nella colonna dell'attributo specificato;
- nel secondo caso, indicando l'asterisco, la funzione **COUNT(*)** calcola il numero delle righe della tabella, incluse quelle con campi di tipo **NULL**.

La funzione calcola solo il numero delle righe, indipendentemente dai valori in esse memorizzati.

Se si utilizza una selezione (clausola **WHERE**), la funzione **COUNT** restituisce il numero delle righe che soddisfano la condizione specificata.

Esempio: si consideri, ancora una volta, l'esempio indicato in precedenza, reperibile sempre all'url <http://www.riochierego.it/mobile/docs/quinta/lab/ESERCITAZIONE-db-esempio-DIPENDENTE-REPARTO-SOLO-TESTO-QUERY-ALUNNO.pdf>

in cui risulta presente la seguente relazione il cui schema è:

Dipendente (CodD, Cognome, Nome, DataN, Stipendio, Livello, DataA, CodR1)

e che immaginiamo, in un certo istante di tempo t, essere rappresentata sempre da una tabella così popolata:

CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1
D4	Verstappen	Max	1980-11-12	4200.00	9	2019-06-27	NULL

La seguente interrogazione restituisce il numero dei valori non nulli presenti nella colonna (attributo) "CodD" della tabella "Dipendente":

```
SELECT COUNT(CodD)
FROM Dipendente;
```

COUNT(CodD)
4

La seguente interrogazione restituisce il numero dei valori non nulli presenti nella colonna (attributo) "Livello" della tabella "Dipendente":

```
SELECT COUNT(Livello)
FROM Dipendente;
```

COUNT(Livello)
4

La seguente interrogazione restituisce il numero dei valori non nulli e distinti presenti nella colonna (attributo) "Livello" della tabella "Dipendente":

```
SELECT COUNT(DISTINCT Livello) AS NumLivelli
FROM Dipendente;
```

NumLivelli
3

La seguente interrogazione restituisce il numero dei valori non nulli presenti nella colonna (attributo) "CodR1" della tabella "Dipendente":

```
SELECT COUNT(CodR1)
FROM Dipendente;
```

COUNT(CodR1)
3

Per contare tutte le righe della tabella è possibile utilizzare l'asterisco **COUNT(*)** al posto del nome dell'attributo ed in tal caso NON bisogna utilizzare lo specificatore **DISTINCT**

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

COUNT(*)
4

Nota bene: Attenzione ad usare, nella stessa **SELECT**, le funzioni di aggregazione in abbinata alla proiezione di altre colonne della tabella.

Un tale tipo di query, seppur sintatticamente corretta, darebbe informazioni in parte prive di significato.

Infatti la seguente interrogazione oltre al numero di righe presenti nella tabella (informazione significativa), restituisce anche (in questo caso) il cognome ed il nome della prima ennupla presente nella tabella "Dipendente" (informazioni non significative):

```
SELECT Cognome, Nome, COUNT(*)
FROM Dipendente;
```

Cognome	Nome	COUNT(*)
Costanzo	Renato	4

2) Funzione MIN**3) Funzione MAX**SINTASSI:

MIN | MAX([DISTINCT] <NomeAttributo>) [AS <Alias>]

Le funzioni **MIN** e **MAX** possono essere usate su qualsiasi dato ordinabile numerico o alfanumerico, e restituiscono rispettivamente il valore minimo e il valore massimo tra i valori della colonna, anche di tipo carattere, specificata come argomento della funzione.

La seguente interrogazione restituisce il valore minimo della colonna (attributo) "Stipendio" della tabella "Dipendente":

```
SELECT MIN(Stipendio)
FROM Dipendente;
```

MIN(Stipendio)
1200.00

La seguente interrogazione restituisce il valore massimo della colonna (attributo) "CodD" della tabella "Dipendente":

```
SELECT MAX(CodD)
FROM Dipendente;
```

MAX(CodD)
D4

Anche in tal caso, se si utilizza una selezione (clausola **WHERE**), tali funzioni calcolano il valore minimo e massimo dei valori di una colonna considerando solo le righe che soddisfano la condizione.

La seguente interrogazione restituisce il valore minimo della colonna (attributo) "Stipendio" della tabella "Dipendente", relativamente ai soli dipendenti di livello 8:

```
SELECT MIN(Stipendio)
FROM Dipendente
WHERE (Livello = 8);
```

MIN(Stipendio)
1350.00

Le funzioni **MIN** e **MAX** ignorano i campi con valore **NULL** e possono avere come argomento un'espressione anziché il nome di un attributo.

La seguente interrogazione restituisce il massimo degli Stipendi relativi ai dipendenti, diminuito del 20%:

```
SELECT MAX(Stipendio - (Stipendio*20)/100) AS NewMax
FROM Dipendente;
```

NewMax
3360.0000

o più semplicemente

```
SELECT MAX(0.80 * Stipendio) AS NewMax
FROM Dipendente;
```

E' possibile utilizzare anche la parola chiave **DISTINCT**, sebbene in questo caso non abbia molto senso.

4) Funzione SUM

SINTASSI:

```
SUM([DISTINCT] <NomeAttributo> ) [AS <Alias>]
```

La funzione **SUM** può essere usata solo su dati numerici e restituisce la somma di tutti i valori contenuti in una colonna specificata come argomento della funzione.

La seguente interrogazione restituisce la somma degli stipendi relativi a tutte le righe presenti nella tabella "Dipendente":

```
SELECT SUM(Stipendio)
FROM Dipendente;
```

SUM(Stipendio)
9100.00

Nel caso siano presenti valori **NULL**, essi saranno considerati come aventi valore 0.

Se si utilizza una selezione (clausola **WHERE**), la funzione prenderà in esame solo le righe che soddisfano la condizione specificata.

La seguente interrogazione restituisce la somma degli Stipendi relativi ai dipendenti di 8 livello presenti nella tabella "Dipendente"

```
SELECT SUM(Stipendio) AS Totale
FROM Dipendente
WHERE (Livello = 8);
```

Totale
3700.00

Se necessario, è possibile utilizzare anche la parola chiave **DISTINCT**.

L'argomento della funzione **SUM** può anche essere un'espressione numerica contenente i nomi della/e colonna/e di tipo numerico interessata/e.

La seguente interrogazione restituisce la somma degli Stipendi relativi ai dipendenti, aumentata del 20%:

```
SELECT SUM(Stipendio + (Stipendio * 20)/100) AS Totale
FROM Dipendente;
```

Totale
10920.000000

o più semplicemente

```
SELECT SUM(1.20 * Stipendio) AS Totale
FROM Dipendente;
```

5) Funzione **AVG**

SINTASSI:

AVG(**[DISTINCT]** <NomeAttributo>) **[AS <Alias>]**

La funzione **AVG** (dall'inglese Average) **può essere usato solo su dati numerici o intervalli di tempo** e calcola la media aritmetica dei valori numerici contenuti in una determinata colonna di una tabella, con l'eventuale aggiunta dell'opzione **DISTINCT**.

L'argomento della funzione può essere un'espressione aritmetica anziché il nome di un attributo.

La funzione **non include** nel calcolo i valori di tipo **NULL** presenti nella colonna.

La seguente interrogazione restituisce la media degli stipendi (diversi da NULL) relativi a tutte le righe presenti nella tabella "Dipendente":

SELECT AVG (Stipendio)	+-----+
FROM Dipendente;	AVG(Stipendio)
	+-----+
	2275.000000
	+-----+

GLI ORDINAMENTI

Finora non abbiamo fatto alcuna ipotesi **sull'ordine** in cui possono apparire le righe di una tabella risultato dell'istruzione **SELECT** (ossia di una query).

In SQL è possibile ordinare tali righe utilizzando alcune clausole all'interno dell'istruzione **SELECT**

ORDER BY <NomeAttributo1> **[ASC | DESC]**, ... , <NomeAttributoN> **[ASC | DESC]**

dove **ASC** sta per *ordine crescente* (dall'inglese "ascending" ed è l'ordinamento di DEFAULT se non specificato altrimenti) e **DESC** per *ordine decrescente* (dall'inglese "descending")

L'ordinamento viene eseguito dapprima sul primo attributo indicato dall'**ORDER BY** e poi a parità di ordinamento su di esso, si ordina sulla base del secondo attributo e così via.

La seguente interrogazione restituisce i nominativi dei dipendenti che guadagnano più di 3000,00 euro ordinati per cognome decrescente:

SELECT Cognome, Nome	+-----+
FROM Dipendente	Cognome Nome
WHERE (Stipendio > 2000.00)	+-----+
ORDER BY Cognome DESC ;	Verstappen Max
	Costanzo Maurizio
	+-----+

La seguente interrogazione invece restituisce i nominativi di tutti i dipendenti ordinati per cognome e nome crescenti:

SELECT Cognome, Nome	+-----+
FROM Dipendente	Cognome Nome
ORDER BY Cognome, Nome;	+-----+
<i>equivalente a</i>	Costanzo Maurizio
SELECT Cognome, Nome	Costanzo Renato
FROM Dipendente	Marini Valeria
ORDER BY Cognome ASC , Nome ASC ;	Verstappen Max
	+-----+

I RAGGRUPPAMENTI

Capita con una notevole frequenza di dover effettuare calcoli o statistiche su dati memorizzati, che abbiano qualche caratteristica in comune.

SQL mette a disposizione, a questo scopo, un potente strumento: i **raggruppamenti**.

“Raggruppare” significa “mettere insieme” tutti i dati, accomunati da qualche caratteristica, su cui vanno fatti alcuni tipi di conteggi.

Le funzioni di aggregazione che, come abbiamo visto in precedenza, permettono di ottenere informazioni riepilogative di sintesi delle ennuple di una tabella, sono in genere abbinate alle **clausole di raggruppamento** all’interno dell’istruzione **SELECT**.

Per esempio si determina con una query di raggruppamento l'individuazione dello stipendio minimo o massimo dei dipendenti di un determinato reparto (codice); oppure la somma degli stipendi dei dipendenti con un determinato livello e via dicendo

Per poter eseguire un raggruppamento all’interno di una **SELECT**, occorre rispettare la seguente SINTASSI:

GROUP BY <NomeAttributo1>, ..., <NomeAttributoN> [**HAVING** <CondizioneGruppo>]

Con l’aggiunta della clausola di raggruppamento il significato della **SELECT** è il seguente:

- viene eseguito, se necessario, il prodotto delle tabelle presenti nella clausola **FROM**;
- su tale prodotto, se previsto, si effettua una restrizione in base alla clausola **WHERE**;
- la tabella risultante viene logicamente partizionata in gruppi di righe in base alla clausola **GROUP BY**. (due righe appartengono allo stesso gruppo se hanno gli stessi valori per gli attributi elencati nella clausola GROUP BY);
- tutti i gruppi che non soddisfano l’eventuale clausola **HAVING** vengono ora eliminati.

*N.B. Ricordiamo che con la parola chiave **DISTINCT**, già presentata in precedenza, dopo il comando parola **SELECT**, abbiamo visto in pratica la forma più semplice di aggregazione: quella che distingue le righe uguali riportandole quindi una sola volta.*

SELECT DISTINCT Livello
FROM Dipendente;

SELECT Livello
FROM Dipendente
GROUP BY Livello;

```
+-----+
| Livello |
+-----+
|       7 |
|       8 |
|       9 |
+-----+
```

Da quanto detto, possiamo desumere che se viene specificata una clausola **GROUP BY**, allora nella clausola **SELECT** dovrà essere presente:

- o un campo specificato nella clausola **GROUP BY**;
- oppure una **funzione di aggregazione**.

Questo perché quando il motore aggrega le righe deve sapere come comportarsi per ogni campo da restituire.

Per raggruppare i dipendenti in base al loro livello e conoscere lo stipendio medio per livello possiamo scrivere:

```
SELECT Livello, AVG(Stipendio)
FROM Dipendente
GROUP BY Livello;
```

Livello	AVG(Stipendio)
7	1200.000000
8	1850.000000
9	4200.000000

Filtraggio sul raggruppamento

A differenza di WHERE, che agisce a livello di singola riga, la parola chiave opzionale **HAVING** permette di effettuare un filtraggio sul raggruppamento.

Questa clausola si inserisce subito dopo la GROUP BY.

Il criterio di filtraggio può contenere una qualsiasi funzione di raggruppamento ma anche un altro eventuale predicato logico:

Per raggruppare i dipendenti in base al loro livello e conoscere lo stipendio medio a partire dall'8° livello, possiamo scrivere:

```
SELECT Livello, AVG(Stipendio)
FROM Dipendente
GROUP BY Livello
HAVING Livello >= 8 ;
```

Livello	AVG(Stipendio)
8	1850.000000
9	4200.000000

Del tutto analoga, in questo caso, la seguente query che non fa utilizzo della clausola HAVING

```
SELECT Livello, AVG(Stipendio)
FROM Dipendente
WHERE (Livello >= 8)
GROUP BY Livello;
```

Nota bene

Nell'istruzione **SELECT**, la differenza principale tra la clausola opzionale **WHERE** (detta "taglio orizzontale") e la clausola opzionale **HAVING** sui raggruppamenti (detto "taglio orizzontale sui gruppi") è che la clausola **WHERE** viene utilizzata per filtrare i record (ossia le singole ennuple) **PRIMA** che si verifichi un raggruppamento o un'aggregazione, mentre **HAVING** si utilizza per filtrare i record **DOPO** un raggruppamento o che si è verificata un'aggregazione.

```
SELECT Livello, COUNT(*)
FROM Dipendente
GROUP BY Livello
HAVING COUNT(*) = 2;
```

analoga a

```
SELECT Livello, COUNT(*) AS Totale
FROM Dipendente
GROUP BY Livello
HAVING Totale = 2;
```

```
SELECT Livello, COUNT(*)
FROM Dipendente
WHERE COUNT(*) = 2
GROUP BY Livello;
```

ERRORE!!!

Non è possibile utilizzare le funzioni di aggregazione nella clausola WHERE a meno che esse non si trovino in una sottoquery annidata

Linguaggio SQL: WHERE vs HAVING	
Il WHERE è una clausola del linguaggio SQL che viene utilizzata per specificare una condizione durante il recupero dei dati da una singola tabella o dall'unione di più tabelle.	L'HAVING è una clausola del linguaggio SQL che specifica che un'istruzione di SELECT deve restituire solo le righe in cui i valori raggruppati soddisfano le condizioni specificate.
Scopo	
La clausola WHERE viene utilizzata per esprimere condizioni sui valori delle ennuple ossia per filtrare le righe di una singola tabella o di più di più tabelle	La clausola HAVING viene utilizzata per esprimere condizioni su valori di gruppi di ennuple aggregate ossia per filtrare i gruppi ottenuti dopo una GROUP BY
Aggregazione	
La clausola WHERE non può essere utilizzata con le funzioni di aggregazione, a meno che esse non si trovino in una sottoquery annidata.	Le funzioni di aggregazione possono essere tranquillamente utilizzate con la clausola HAVING .
Metodo di filtraggio	
La clausola WHERE si comporta come un PRE-filtro (prima del raggruppamento)	La clausola HAVING si comporta come un POST-filtro (dopo il raggruppamento)
Raggruppa per ordine	
La clausola WHERE viene utilizzata PRIMA della clausola GROUP BY	La clausola HAVING viene utilizzata DOPO la clausola GROUP BY .
Usato con	
La clausola WHERE può essere utilizzata con SELECT , UPDATE e DELETE .	La clausola HAVING viene utilizzata solo all'interno di una SELECT .

QUERY ANNIDATE o SUBQUERY

Per rispondere ad **interrogazioni o query** complesse è possibile strutturare più SELECT. Questo permette di costruire una interrogazione al cui interno sono presenti altre interrogazioni dette **sottointerrogazioni o subquery**.

Una sottoquery può essere composta a sua volta da altre query.

Si viene a creare così una struttura di **query annidate**.

Per eseguire sottoquery annidate si deve eseguire prima l'interrogazione più interna e poi eseguire quella più esterna fino ad arrivare alla query principale.

In una interrogazione che ne richiama un'altra al suo interno possiamo distinguere:

- la **query principale** o **query esterna**: quella individuata dalla prima parola chiave SELECT incontrata;
- la **query secondaria** o **sottoquery** o **subquery** o **query interna**: quella individuata dalla seconda parola chiave SELECT incontrata (delimitata tra parentesi tonde).

La sottoquery genera una tabella che si chiama *tabella derivata* che può essere composta da:

- a) **un solo valore** (tabella formata da una sola riga e da una sola colonna) ed in questo caso si parla di *tabella o query scalare*;
- b) **una sola riga (ma più colonne);**
- c) **più righe e più colonne.**

} in generale, più valori

Una tabella scalare si può utilizzare tutte le volte in cui è richiesto l'utilizzo di un singolo valore nella query. E' spesso conveniente utilizzare nelle sottoquery la clausola **AS** per assegnare un nome alla tabella ottenuta

SOTTOINTERROGAZIONI che producono un solo valore

Quando si è sicuri che una sottoquery produce un solo valore come risultato è possibile utilizzare tale sottointerrogazione nelle espressioni della query principale.

```
SELECT <Lista Attributi>
FROM <ListaTabelle>
WHERE <Attributo> <OperatoreDiConfronto> (<SubQuery>);
```

dove <OperatoreDiConfronto> è uno tra <, >, ≤, ≥, =, <>.

Esempio: consideriamo le seguenti due relazioni

Regista (CodRegista, Cognome, Nome, Compenso)

Film (CodFilm, Titolo, Budget, CodRegia)

con l'attributo "CodRegia" che è chiave esterna (FK) sull'attributo "CodRegista" della relazione "Regista"

Supponiamo di volere effettuare la seguente interrogazione

Query: Mostrare il cognome ed il nome del regista del film "Via col vento"

```
SELECT Regista.Cognome, Regista.Nome
FROM Regista, Film
WHERE (Regista.CodRegista = Film.CodRegia) AND (Film.Titolo = "Via con vento");
```

analogo alla **SOTTOINTERROGAZIONE o SUBQUERY**

```
SELECT Regista.Cognome, Regista.Nome
FROM Regista
WHERE CodRegista = (SELECT CodRegia
FROM Film
WHERE Titolo = "Via col vento");
```

Esempio: si consideri, ancora una volta, l'esempio discusso a lezione, reperibile all'url

<http://www.riochierego.it/mobile/docs/quinta/lab/ESERCITAZIONE-db-esempio-DIPENDENTE-REPARTO-SOLO-TESTO-QUERY-ALUNNO.pdf>

in cui risulta presente la seguente relazione il cui schema è:

Dipendente (CodD, Cognome, Nome, DataN, Stipendio, Livello, DataA, CodR1)

e che immaginiamo, in un certo istante di tempo t , essere rappresentata da una tabella così popolata:

CodD	Cognome	Nome	DataN	Stipendio	Livello	DataA	CodR1
D1	Costanzo	Renato	1950-08-25	1200.00	7	2017-11-15	R1
D2	Marini	Valeria	1965-11-09	1350.00	8	2018-10-24	R2
D3	Costanzo	Maurizio	1975-03-13	2350.00	8	2016-08-08	R1
D4	Verstappen	Max	1980-11-12	4200.00	9	2019-06-27	NULL

Query: Mostrare il cognome ed il nome del dipendente che guadagna di più

N.B. Si ricordi che nell'utilizzo nell'ambiente **MySQL** è **sintatticamente errato** lasciare uno spazio tra il nome della funzione di aggregazione e la prima parentesi tonda aperta!

```
SELECT Cognome, Nome
FROM Dipendente
WHERE Stipendio = (SELECT MAX(Stipendio)
                   FROM Dipendente);
```

```
+-----+-----+
| Cognome | Nome |
+-----+-----+
| Verstappen | Max |
+-----+-----+
```

Query: Mostrare il cognome ed il nome dei dipendenti che guadagnano più della media degli stipendi erogati dall'azienda

```
SELECT Cognome, Nome
FROM Dipendente
WHERE Stipendio > (SELECT AVG(Stipendio)
                   FROM Dipendente);
```

```
+-----+-----+
| Cognome | Nome |
+-----+-----+
| Costanzo | Maurizio |
| Verstappen | Max |
+-----+-----+
```

Query: Mostrare il cognome ed il nome del dipendente più anziano

```
SELECT Cognome, Nome
FROM Dipendente
WHERE DataN = (SELECT MIN(DataN)
               FROM Dipendente);
```

```
+-----+-----+
| Cognome | Nome |
+-----+-----+
| Costanzo | Renato |
+-----+-----+
```

Query: Mostrare il cognome ed il nome del dipendente più giovane

```
SELECT Cognome, Nome
FROM Dipendente
WHERE DataN = (SELECT MAX(DataN)
               FROM Dipendente);
```

```
+-----+-----+
| Cognome | Nome |
+-----+-----+
| Verstappen | Max |
+-----+-----+
```

SOTTOINTERROGAZIONI (SUBQUERY) che producono valori appartenenti ad un insieme

Nella costruzione delle **sottointerrogazione (o sottoquery o subquery)** è possibile utilizzare i seguenti predicati per effettuare ricerche sui valori di attributi che soddisfano proprietà di appartenenza a insiemi di valori.

I possibili predicati che possiamo utilizzare sono:

- (1) **ANY** e **ALL**;
- (2) **IN** e **NOT IN**;
- (3) **EXISTS** e **NOT EXISTS**

(1) **ANY** e **ALL** vengono utilizzati nelle condizioni delle query del tipo

```
SELECT <Lista Attributi>
FROM <ListaTabelle>
WHERE <Attributo> <OperatoreDiConfronto> ANY | ALL (< SubQuery >);
```

dove <OperatoreDiConfronto> è uno tra <, >, ≤, ≥, =, <>, ed

- **ANY** indica che la condizione della clausola WHERE è vera se il valore dell'attributo <Attributo> compare **in almeno uno** dei valori restituiti dalla sottoquery < SubQuery >
- **ALL** indica che la condizione della clausola WHERE è vera se il valore dell'attributo <Attributo> compare **in tutti** quelli restituiti dalla sottoquery < SubQuery >

N.B. Ovviamente la sottoquery < SubQuery > prevede la selezione su un unico attributo

Esempio: Consideriamo le seguenti relazioni

Regista (CodRegista, Cognome, Nome)

Attore (CodAttore, Cognome, Nome)

*Per ottenere i registi che sono anche attori scriveremo (quindi **Regista** ∩ **Attore**)*

Nel caso non funzioni l'operatore **INTERSECT** (caso MySql) non resta che intraprendere la strada di una query annidata (subquery) in grado di fornire gli stessi risultati (logicamente equivalente):

```
(SELECT Cognome, Nome
FROM Regista)
INTERSECT
(SELECT Cognome, Nome
FROM Attore);
```

```
#alternativa con SUBQUERY (Occhio questa query controlla solo il Cognome, non il Nome)
SELECT Cognome, Nome
FROM Regista
WHERE Cognome = ANY (SELECT Cognome
FROM Attore);
```

*Per ottenere i registi che **NON** sono anche attori scriveremo (quindi **Regista** - **Attore**)*

Nel caso non funzioni l'operatore **MINUS** (caso MySql) non resta che intraprendere la strada di una query annidata (subquery) in grado di fornire gli stessi risultati (logicamente equivalente):

```
(SELECT Cognome, Nome
FROM Regista)
MINUS
(SELECT Cognome, Nome
FROM Attore);
```

```
#alternativa con SUBQUERY (Occhio questa query controlla solo il Cognome, non il Nome)
SELECT Cognome, Nome
FROM Regista
WHERE Cognome <> ALL (SELECT Cognome
FROM Attore);
```

(2) **IN** e **NOT IN** vengono utilizzati nelle condizioni delle query del tipo

```
SELECT <Lista Attributi>
FROM <ListaTabelle>
WHERE <Attributo> IN | NOT IN (< SubQuery >);
```

- **IN**: la condizione della clausola WHERE è vera se il valore dell'attributo <Attributo> **appartiene** all'insieme dei valori forniti dalla sottoquery < SubQuery >

(N.B. il predicato **IN** equivale a **= ANY**);

- **NOT IN**: la condizione della clausola WHERE è vera se il valore dell'attributo <Attributo> **non appartiene** all'insieme dei valori forniti dalla sottoquery < SubQuery >.

(N.B. il predicato **NOT IN** equivale a **≠ ALL**);

N.B. Ovviamente la sottoquery < SubQuery > prevede la selezione su un unico attributo

Esempio: Consideriamo ancora una volta le seguenti relazioni

Regista (CodRegista, Cognome, Nome)

Attore (CodAttore, Cognome, Nome)

*Per ottenere i registi che sono anche attori scriveremo (quindi **Regista** ∩ **Attore**)*

Nel caso non funzioni l'operatore **INTERSECT** (caso MySql) non resta che intraprendere la strada di una query annidata (subquery) in grado di fornire gli stessi risultati (logicamente equivalente):

```
(SELECT Cognome, Nome
FROM Regista)
INTERSECT
(SELECT Cognome, Nome
FROM Attore);
```

```
#alternativa con SUBQUERY (Occhio questa query controlla solo il Cognome, non il Nome)
SELECT Cognome, Nome
FROM Regista
WHERE Cognome IN (SELECT Cognome
FROM Attore);
```

*Per ottenere i registi che **NON** sono anche attori scriveremo (quindi **Regista** - **Attore**)*

Nel caso non funzioni l'operatore **MINUS** (caso MySql) non resta che intraprendere la strada di una query annidata (subquery) in grado di fornire gli stessi risultati (logicamente equivalente):

```
(SELECT Cognome, Nome
FROM Regista)
MINUS
(SELECT Cognome, Nome
FROM Attore);
```

```
#alternativa con SUBQUERY (Occhio questa query controlla solo il Cognome, non il Nome)
SELECT Cognome, Nome
FROM Regista
WHERE Cognome NOT IN (SELECT Cognome
FROM Attore);
```

(3) **EXISTS** e **NOT EXISTS** vengono utilizzati nelle condizioni delle query del tipo

```
SELECT <Lista Attributi>
FROM <ListaTabelle>
WHERE EXISTS | NOT EXISTS (< SubQuery >);
```

dove la condizione:

- **EXISTS**: la condizione della clausola **WHERE** è vera se la sottoquery < SubQuery > produce una **tabella non vuota**;

- **NOT EXISTS**: la condizione della clausola **WHERE** è vera se la sottoquery < SubQuery > produce una **tabella vuota** (senza alcuna riga).

N.B. La sottoquery < SubQuery > può prevedere la presenza di più righe e colonne

Esempio: si consideri, ancora una volta, l'esempio discusso a lezione, reperibile all'url

<http://www.riochierego.it/mobile/docs/quinta/lab/ESERCITAZIONE-db-esempio-CON-STORICO-FORNITORE-FORNISCE-ARTICOLO-SOLO-TESTO-QUERY-ALUNNO.pdf>

in cui risultano presenti le seguenti relazioni:

Fornitore (CodF, Cognome, Nome, DataN)

Articolo (CodA, Descrizione, Prezzo)

Fornisce (IdF, CodF1, CodA1, DataF, Qta)

con l'attributo "CodF1" della relazione "Fornisce" che risulta essere chiave esterna o foreign key (o FK) sull'attributo "CodF" della relazione "Fornitore"

con l'attributo "CodA1" della relazione "Fornisce" che risulta essere chiave esterna o foreign key (o FK) sull'attributo "CodA" della relazione "Articolo"

Query: Visualizzare tutti i fornitori che forniscono un qualche articolo

Una prima soluzione consisterebbe nell'effettuare un **EQUI-JOIN** tra le tabelle **Fornitore** e **Fornisce** eliminando le eventuali ennuple duplicate in quanto occorre tenere presente che un fornitore può fornire in generale più articoli. Quindi:

```
SELECT DISTINCT Fornitore.*
FROM Fornitore, Fornisce
WHERE (Fornitore.CodF = Fornisce.CodF1);
```

Output della query

CodF	Cognome	Nome	DataN
F_01	ROSSI	MARIO	1965-01-01
F_02	GIALLI	ANDREA	1960-08-08
F_03	ROSSI	ELIA	1963-07-03

Una soluzione alternativa è offerta proprio dall'uso dell'operatore **EXISTS** posto all'interno della seguente query annidata:

```
SELECT Fornitore.*
FROM Fornitore
WHERE EXISTS (SELECT *
              FROM Fornisce
              WHERE Fornitore.CodF = Fornisce.CodF1);
```

Anche se l'uso di **EXISTS** è facilmente comprensibile, ciò che può risultare meno intuitivo è il modo in cui la **subquery** è correlata con la **query principale**: all'interno della clausola **WHERE**, infatti, abbiamo utilizzato il campo *CodF* della tabella *Fornitore*.

Ciò implica che la subquery dovrà essere eseguita con un parametro diverso per ogni riga estratta dalla query esterna.

Il risolto pratico è un significato degrado delle prestazioni: in questi casi è sempre meglio preferire soluzioni alternative, considerato come abbiamo visto anche in questo caso, che le subquery correlate possono essere trasformate in operazioni di **EQUI-JOIN** o simili.

Query: Visualizzare tutti i fornitori che NON forniscono alcun articolo

Analizziamo dapprima due soluzioni con query annidate e con gli operatori **ALL** e **NOT IN** di cui abbiamo già, parlato in precedenza

```
SELECT Fornitore.*
FROM Fornitore
WHERE CodF <> ALL (SELECT DISTINCT CodF1
                   FROM Fornisce);
```

Output della query

CodF	Cognome	Nome	DataN
F_04	BIANCHI	VINCENZO	1968-11-24

```
SELECT Fornitore.*
FROM Fornitore
WHERE CodF NOT IN (SELECT DISTINCT CodF1
                  FROM Fornisce);
```

Una soluzione alternativa, più elegante delle precedenti, è offerta dall'uso dell'operatore **NOT EXISTS** posto all'interno della seguente query annidata:

```
SELECT Fornitore.*
FROM Fornitore
WHERE NOT EXISTS (SELECT *
                  FROM Fornisce
                  WHERE Fornitore.CodF = Fornisce.CodF1);
```

Anche se l'uso di **NOT EXISTS** è facilmente comprensibile, ciò che può risultare meno intuitivo è il modo in cui la **subquery** è correlata con la query principale: all'interno della clausola **WHERE**, infatti, abbiamo utilizzato il campo *CodF* della tabella *Fornitore*.

Ciò implica che la subquery dovrà essere eseguita con un parametro diverso per ogni riga estratta dalla query esterna.

Il risvolto pratico è un significato degrado delle prestazioni: in questi casi è sempre meglio preferire soluzioni alternative, considerato come abbiamo visto anche in questo caso, che le subquery correlate possono essere trasformate con subquery più "performanti" (in questo caso **ALL** e **NOT IN**).

UTILIZZO di Subquery nella clausola FROM

E' possibile utilizzare una **sottointerrogazione (o sottoquery o subquery)** anche nella clausola **FROM**, con questa sintassi:

```
SELECT .....
FROM (<SubQuery>) [AS] nome ...
```

Notate che è obbligatorio assegnare un **nome** alla subquery, per poterla referenziare nelle altre parti della query.

Ad esempio:

```
SELECT t1.*, Tabella2.*
FROM (SELECT A1, A2
      FROM Tabella1
      WHERE A3 > 5) AS t1, Tabella2
WHERE t1.K1 = Tabella2.K2;
```

In questo caso l'output della subquery viene chiamato "**t1**" ed il riferimento è usato sia nella **SELECT** sia nella condizione di **EQUI-JOIN**.

Esempio di sottoquery che produce una tabella derivata

Consideriamo le seguenti tre relazioni relative all'utilizzo di laboratori da parte di una classe di studenti:

Laboratorio (CodLab, Denominazione, NumPosti,)

Classe (CodClasse, NumAlunni)

Utilizza (CodClasse, CodLab,)

con l'attributo "CodClasse" che è chiave esterna (FK) sull'attributo "CodClasse" della relazione "Classe"
con l'attributo "CodLab" che è chiave esterna (FK) sull'attributo "CodLab" della relazione "Laboratorio"

Consideriamo la seguente interrogazione:

Query: Vogliamo conoscere la denominazione dei laboratori utilizzati dalla classe "A45"

Partendo dall'**EQUI-JOIN** tra le tabelle **Laboratorio** e **Utilizza**

```
SELECT Laboratorio.Denominazione
FROM Laboratorio, Utilizza
WHERE (Laboratorio.CodClasse = Utilizza.CodClasse);
```

è possibile, anche per ragioni di efficienza, "anticipare" la selezione per **CodClasse = "A45"** sulla tabella "Utilizza" prima di effettuare il join giungendo alla seguente subquery:

```
SELECT Denominazione
FROM Laboratorio,
      (SELECT CodLab
       FROM Utilizza
       WHERE CodClasse = "A45") AS t1
WHERE Laboratorio.CodLab = t1.CodLab ;
```

ISTRUZIONI DCL del linguaggio SQL

In informatica il **Data Control Language (DCL)** è un linguaggio utilizzato nel SQL per fornire o revocare agli utenti i permessi necessari per poter utilizzare i comandi di Data Definition Language (DDL) e Data Manipulation Language, oltre agli stessi comandi DCL (che gli servono a sua volta per poter modificare i permessi su alcuni oggetti).

Lo standard SQL non specifica in quale modo debba essere possibile creare, eliminare o modificare gli utenti di un database.

La maggior parte dei DBMS a tale scopo implementa i comandi non standard **CREATE USER** (che crea un utente e specifica quali permessi deve avere) e **DROP USER** (che elimina un utente).

Per la modifica, alcuni DBMS implementano **ALTER USER**, ma non è un comando molto diffuso, mentre è più diffuso SET PASSWORD.

In sintesi l'istruzione per creare un utente è:

```
CREATE USER <username>@<host>
[ IDENTIFIED BY <password> ] ;
```

dove

- **<username>**: Specifica il nome dell'utente che vogliamo creare o al quale vogliamo assegnare nuovi diritti o privilegi
- **<host>**: Specifica il/gli host da cui è ammessa la connessione. Se impostato a '%' vuol dire che l'utente potrà connettersi al database da qualsiasi host, senza limitazione.
- **<password>**: Specifica la password (scritta tra apici singoli o doppi apici) associata all'utente che stiamo creando.

N.B. La password viene scritta "in chiaro" ma viaggia sulla rete in forma criptata da MySQL tramite la funzione PASSWORD(). Se si vuole visualizzare il valore criptato corrispondente alla chiave scelta, basta effettuare una delle due query seguenti

```
SELECT User, Host, Password          SELECT PASSWORD(<password-in-chiaro>);
FROM mysql.user
WHERE User = <username>;
```

L'istruzione per eliminare un utente è:

```
DROP USER <username>@<host> ;
```

Esempi di utilizzo:

```
CREATE USER davide@localhost;
CREATE USER davide@localhost IDENTIFIED BY 'pippo';
DROP USER davide@localhost;
```

Una volta creato lo schema relazionale tramite apposite istruzioni che appartengono alla parte **DDL** dell'SQL è possibile impostare le politiche relative alla **sicurezza** dei dati

Quando si parla di **sicurezza dei dati** occorre distinguere i seguenti aspetti:

- sicurezza da **guasti hardware** e *software*;
- sicurezza da **accessi non autorizzati**: *per proteggersi da questa eventualità è necessario:*
 - a) stabilire i **diritti di accesso** ai dati;
 - b) stabilire le **viste** ossia le modalità con le quali gli utenti possono vedere la base dei dati
 - c) guardarsi da tecniche di hacking quali la **SQL INJECTION** (che vedremo poi in dettaglio <http://www.riochierego.it/mobile/docs/quinta/lab/PPT-9-SQL-INJECTION.pdf>)
- sicurezza nelle **transazioni** per ottenere la quale occorre preservare **integrità** e **consistenza** dei dati.

N.B. Si fa presente che avremmo in realtà GIA' dovuto parlare di **VISTE quando abbiamo dettagliato i comandi DDL del linguaggio SQL, in quanto, come vedremo, esse non sono altro che TABELLE SPECIALI.**

La ragione per cui se ne parla diffusamente in questa sezione è che un loro corretto utilizzo, da sole o in aggiunta con alcuni dei privilegi di accesso che si possono assegnare ad un utente, permette l'innalzamento dei livelli di sicurezza di un database.

a) Diritti di accesso ai dati

L'amministratore della base di dati o il **proprietario** delle tabelle create, possono assegnare diversi **diritti di accesso** agli utenti o a gruppi di utenti che dovranno interagire con tali tabelle

I permessi di accesso possono essere assegnati, solitamente dal DBA, tramite l'istruzione SQL **GRANT** la cui sintassi è la seguente:

```
GRANT <ElencoPrivilegi>  
ON <NomeDB>.<NomeTabella>  
TO <username>@<host>  
[ IDENTIFIED BY<password> ]  
[ WITH GRANT OPTION ] ;
```

Come vedete la sintassi sopra proposta è molto semplice ed i singoli campi significano:

- **<ElencoPrivilegi>**: E' una lista di istruzioni di SQL che si vogliono permettere all'utente (CREATE, SELECT, UPDATE, DELETE, ALTER, EXECUTE, ecc.). Se si vuole dare all'utente permessi completi si può utilizzare la parola chiave **ALL**;
- **<NomeDB>**: E' il nome del database che stiamo prendendo in considerazione. Per tutti gli altri database non varranno le regole che stiamo specificando. Se si vuole fare riferimento a tutti i database presenti si può utilizzare il carattere asterisco (*);
- **<NomeTabella>**: Inserendo il nome di una tabella, si fa riferimento solo ad essa. Per tutte le altre tabelle non varranno le regole che stiamo specificando. Se si vuole fare riferimento a tutte le tabelle si può utilizzare il carattere asterisco (*);
- **<username>**: Specifica il nome dell'utente già esistente al quale vogliamo assegnare nuovi diritti o privilegi **ma se l'utente non esiste, la riga relativa viene aggiunta alla tabella utenti e 'password' sarà la sua password. Se l'utente esiste già e la password specificata questa viene sostituita altrimenti se la password non è stata specificata, non verrà modificata ;**
- **<host>**: Specifica il/gli host da cui è ammessa la connessione. Se impostato a '%' vuol dire che l'utente potrà connettersi al database da qualsiasi host, senza limitazione;
- **<password>**: Specifica la password (scritta tra apici singoli o doppi apici) associata all'utente che stiamo creando. La password va scritta "in chiaro". Se si desidera inserire la password in forma criptata tramite la funzione PASSWORD() di MySQL, si deve far precedere la stringa criptata dalla parola **PASSWORD**.

Nota Bene: Ad un certo utente può essere assegnato, oltre a tutti gli altri privilegi previsti nella clausola ALL, anche quello di GRANT, che gli permetterà di assegnare ad altri utenti i propri permessi.

Se ciò accade l'utente potrà assegnare ad altri utenti tutti i propri permessi: non solo quelli ricevuti con questa istruzione, ma anche quelli che aveva già e quelli che riceverà in futuro.

L'istruzione SQL **REVOKE** revoca i privilegi ad un utente del database nel compiere determinate azioni precedentemente assegnate tramite l'istruzione **GRANT**. La sua sintassi è la seguente

```
REVOKE <ElencoPrivilegi> oppure GRANT OPTION  
ON <NomeDB>.<NomeTabella>  
FROM <user>@<host>;
```

N.B. Il privilegio di GRANT va revocato ad un utente cui è stato precedentemente assegnato, con un'istruzione a parte rispetto alla revoca di tutti gli altri privilegi possibili

Il DBA può revocare i diritti di accesso ad un utente anche senza l'utilizzo della sua password.

N.B. Per controllare i privilegi assegnati ad un determinato utente possiamo utilizzare

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

oppure, nel caso sia quello con il quale siamo loggati,

```
SHOW GRANTS FOR CURRENT_USER;
```

b) Le VISTE

Una **Vista o View** è una **relazione (tabella) virtuale** ottenuta attraverso un processo di interrogazione di una o più relazioni (tabelle) fisicamente memorizzate sul database (generate attraverso un'istruzione precedente di **CREATE TABLE**).

Una **Vista o View** viene quindi creata tramite un'operazione di **“mapping”** (mappatura) sulle relazioni (tabelle) effettivamente memorizzate nel database.

Le istanze di una **Vista** sono il risultato di una query che viene valutata **dinamicamente** ogni volta che si fa riferimento alla vista stessa.

Dopo essere stata creata, una **vista** può essere facilmente **interrogata** tramite il comando **SELECT**

Inoltre una vista può essere **aggiornabile o “updatable”** (ossia è possibile operare su di essa i comandi di **UPDATE** e **DELETE**) o **inseribile o “insertable”** (ossia è possibile operare su di essa l'operazione di **INSERT**) come una qualunque altra tabella del database.

Non tutte le viste sono aggiornabili e scrivibili.

*N.B. Ad esempio non è “insertable” una vista che mostra i valori massimi di una certa tabella o che raggruppa i record con una clausola **GROUP BY**, perché tali dati non sono fisicamente scritti su una qualche tabella, ma rielaborati tramite una query.*

Se anche tutti i dati contenuti nella vista sono scritti fisicamente nelle tabelle, qualora la vista coinvolga diverse tabelle, è necessario che tra esse vi sia una relazione uno a uno.

*Inoltre generalmente non è modificabile una vista basata su una **UNION**.*

Nota Bene: Ogni inserimento/modifica/ cancellazione apportata su di una tabella di tipo vista si ripercuote sulla/e tabella/e fisica/fisiche dalla/e quale/i è stata tratta.

Vantaggi nell'utilizzo delle viste

Uno dei vantaggi principali connessi all'utilizzo delle **viste** sta nel fatto che grazie ad esse è possibile aumentare il grado di sicurezza dei dati nel momento in cui vengono manipolati.

Infatti le **viste** sono in grado di fornire ad un gruppo di utenti **una versione semplificata o parziale** di una realtà che può essere molto più complessa.

Categorie diverse di utenti possono interagire con la base di dati utilizzando il loro punto di vista e trascurando quelli degli altri.

Quindi, in pratica, la creazione di una **vista** ci permette di imporre delle restrizioni sull'accesso ai dati e di limitare in questo modo l'azione delle interrogazioni soltanto a specifiche colonne di una tabella; lo stesso discorso può essere fatto per quanto riguarda le possibilità di accesso ai dati.

N.B. Anche sulle viste **inoltre** è possibile utilizzare l'istruzione **GRANT**.

Quindi è possibile creare utenti con diritti di accesso specifici su tabelle di tipo vista aumentando di fatto il livello di sicurezza nell'accesso ai dati.

Per **creare una vista** si utilizza l'istruzione SQL **CREATE VIEW** la cui sintassi è la seguente:

```
CREATE VIEW <NomeTabellaVista> [<ListaColonne>] AS <QuerySelect> [WITH CHECK OPTION] ;
```

Dove:

- <NomeTabellaVista> è il nome che desideriamo assegnare alla **vista**;
- <ListaColonne> contiene l'elenco delle colonne che parteciperanno alla creazione della **vista**;
N.B. questa voce deve comprendere lo stesso numero di colonne che viene specificato nella parte relativa a <QuerySelect>; nel caso in cui la voce <ListaColonne> dovesse mancare CREATE VIEW farà riferimento a quelle indicate tramite <QuerySelect>;
- <QuerySelect> è una normale query formulata con l'istruzione **SELECT** che definisce la **vista**;
- **WITH CHECK OPTION** è una clausola che viene applicata ai comandi per la modifica dei dati come INSERT e UPDATE per rispettare la definizione della **vista** apportata tramite <QuerySelect> (*siano, cioè, conformi all'eventuale clausola WHERE specificata in sede di definizione della Vista*).

Quando si crea una **vista** è necessario tenere conto di alcune restrizioni:

- È possibile creare **viste** soltanto nel database corrente.
- Il nome della **vista** deve seguire le regole imposte dall'IDENTIFIER del datasource.
- Il nome della **vista** non deve essere lo stesso della tabella di partenza.
- È possibile creare una **vista** solo se la tabella di partenza permette l'esecuzione di query SELECT.
- Non è possibile definire una **vista** utilizzando il comando **SELECT INTO**.
- Non è possibile definire un trigger in una **vista**.
- Non è possibile definire un indice all'interno di una **vista**.
- Non è possibile associare CREATE VIEW ad altri comandi SQL nella stessa istruzione.

Le tabelle virtuali possono essere alterate esattamente come quelle fisiche, a questo scopo esiste il comando ALTER VIEW che modifica la definizione di una **vista**.

Per **modificare una vista** si utilizza l'istruzione **ALTER VIEW** la cui sintassi è la seguente:

```
ALTER VIEW <NomeTabellaVista> [<ListaColonne>] AS <QuerySelect> [WITH CHECK OPTION] ;
```

In questo caso non vi è nulla da segnalare di particolare, la sintassi per l'alterazione delle tabelle si differenzia da quella di creazione essenzialmente per il comando SQL utilizzato che in questo caso è ALTER VIEW.

Per **eliminare una vista** si utilizza l'istruzione SQL **DROP VIEW** la cui sintassi è la seguente:

```
DROP VIEW <NomeTabellaVista> ;
```

SQL EMBEDDED

Finora abbiamo ipotizzato di utilizzare SQL in modalità **stand-alone** ossia abbiamo supposto che le istruzioni SQL venissero scritte attraverso una qualche interfaccia grafica e venissero eseguite dall'interprete SQL:

La modalità **embedded** si riferisce all'uso di istruzioni SQL all'interno di un linguaggio di programmazione detto **linguaggio di programmazione ospite**.

In questo caso oltre alle istruzioni SQL ed alle istruzioni proprie del linguaggio ospite, esiste un terzo tipo di istruzioni che permettono **l'integrazione ed il controllo del flusso** delle istruzioni SQL all'interno del linguaggio ospite.

Il principale problema della coesistenza di SQL con il linguaggio ospite è il **disadattamento di impedenza o impedance mismatch**: questo problema è causato dal fatto che il modello dei dati di SQL è "*set-oriented*" ossia opera su insieme di ennuple, mentre i linguaggi ospite sono "*record-oriented*" ossia operano e si riferiscono ad un solo record alla volta.

Per evitare questo problema si potrebbe pensare di usare un solo linguaggio **ma ciò non è possibile**:

- se si utilizza **solo il linguaggio ospite** si perdono espressività e semplicità rendendo il codice prodotto incomprensibile;
- se si utilizza **solo il linguaggio SQL** (che è diventato sempre più potente e flessibile) non si hanno ancora a disposizione funzionalità tali da scrivere applicazioni complesse (come un'interfaccia utente oppure un sottoprogramma che calcoli il fattoriale).

Occorre quindi utilizzarli entrambi anche perché:

- a) tutti gli attuali **DBMS** hanno l'**SQL** come **DML**;
- b) lo standard **ANSI** ha definito estensioni particolari per molti linguaggi al fine di includere l'**SQL**.

Le istruzioni SQL per interfacciamento variano a seconda che si tratti di:

- linguaggio ospite **procedurale**;
- linguaggio ospite **ad oggetti**.

N.B. Una soluzione al problema del disadattamento d'impedenza nei linguaggi ospite **procedurali** consiste nell'uso dei **cursori** espliciti.

DEF: Un **cursore** è un **indicatore di posizione** che viene utilizzato per muoversi all'interno di una *tabella* risultato di una *query*.

Esso può essere visto come un **puntatore** ad una particolare riga di una tabella, riga che prende il nome di **riga corrente**.

N.B. Una soluzione al problema del disadattamento d'impedenza nei linguaggi ospite ad **oggetti** consiste nell'uso di classi di oggetti appositamente definite per l'interfacciamento con l'**SQL** che utilizzano dei **cursori impliciti**.

SQL DINAMICO

La tecnica dell'**SQL dinamico o dynamic SQL** non prevede l'inserimento *statico* di istruzioni SQL nel codice del programma ospite ma la loro acquisizione *dinamica o a tempo di esecuzione* da qualsiasi dispositivo di input, oppure la loro generazione a programma.

Quindi le istruzioni SQL non sono note più *a tempo di compilazione* come avviene per l'**SQL embedded** ma vengono create o acquisite *a tempo di esecuzione* dal programma ospite.

Con l'SQL dinamico è possibile mantenere indefiniti fino al momento dell'esecuzione del programma:

- il *testo* dell'istruzione SQL;
- il *numero* ed i *tipi* delle variabili ospiti;
- i *riferimenti* agli oggetti della base di dati.;

Il programma del linguaggio ospite deve chiedere all'SQL di prendere una stringa di caratteri appena letta, trasformarla in istruzione SQL ed eseguirla.

Esistono **due istruzioni SQL** che corrispondono a due diverse modalità di esecuzione di una query:

(A) l'**esecuzione immediata** avviene mediante l'istruzione SQL:

```
EXEC SQL EXECUTE IMMEDIATE <Query>
```

dove:

- <Query> è una stringa che rappresenta una query (composta nel programma ospite) oppure una variabile ospite di tipo stringa.

Questa modalità di esecuzione avviene per istruzioni SQL che non richiedono parametri né in ingresso né in uscita come i comandi di inserimento o cancellazione.

(B) l'**esecuzione in due fasi** avviene quando l'istruzione SQL da eseguire utilizza parametri in ingresso o in uscita oppure deve essere eseguita più volte.

Osserviamo due fasi:

(B1) la fase di preparazione: che consiste nell'associare un nome all'istruzione SQL che può contenere parametri in ingresso rappresentati dal carattere '?'

```
EXEC SQL PREPARE <NomeIstruzioneSQL> FROM <IstruzioneSQL>
```

dove:

- <NomeIstruzioneSQL> è il nome dato all'istruzione <IstruzioneSQL> per poter essere richiamata anche in seguito;
- <IstruzioneSQL> è una istruzione SQL che può contenere parametri in ingresso rappresentati dal carattere '?'

(B2) la fase di esecuzione: che consiste nell'eseguire l'istruzione SQL preparata utilizzando l'istruzione SQL

```
EXEC SQL EXECUTE <NomeIstruzioneSQL>  
[INTO <VariabiliRisultato>] [USING <VariabiliParametro>]
```

dove:

- <NomeIstruzioneSQL> è il nome dato all'istruzione <IstruzioneSQL> nella fase di **PREPARE**;
- <VariabiliRisultato> è un elenco di variabili ospiti che conterranno l'eventuale risultato della query: ogni variabile è preceduta da ':'
- <VariabiliParametro> è un elenco di variabili ospiti che conterranno i parametri da usare nella query

Usare MYSQL Vers. 5.6.40 dalla console di comando in modalità interattiva -

DATABASE FornituraNew

relativo all'associazione "Fornisce" tra le entità "Fornitore" ed "Articolo" di molteplicità N:N

<http://www.riochierego.it/mobile/docs/quinta/lab/ESERCITAZIONE-db-esempio-CON-STORICO-FORNITORE-FORNISCE-ARTICOLO-SOLO-TESTO-QUERY-ALUNNO.pdf>

Se si volesse far rispettare a MySQL le **maiuscole/minuscole** per i nomi dei database e/o delle tabelle e/o degli attributi etc. occorre inserire nel suo file di configurazione **my.ini** nella sezione **[mysqld]** la seguente linea:

lower_case_table_names = 2

e poi, dopo averlo stoppato, far ripartire il servizio

1) loggarsi al DBMS come "root" (DATABASE Administrator di default)

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

dopo di che verrà richiesta con "Enter password" ed occorrerà digitare la password.

Nel nostro caso, scrivere "**root**" (verranno visualizzati **4 asterischi**)

2) creiamo un nuovo utente

```
CREATE USER pippo@localhost IDENTIFIED BY "pippo";
```

N.B. per cancellarlo il DBA userà il comando

```
DROP USER pippo@localhost;
```

3) guardiamo i db creati in precedenza attualmente presenti

```
SHOW DATABASES;
```

4) creiamo come DBA il database "fornituranew" sul quale opereremo

```
CREATE DATABASE FornituraNew;
```

5) selezioniamo il database come DBA

```
USE FornituraNew;
```

6) assegniamo tutti i diritti all'utente "pippo" sul db in oggetto (su tutte le sue tabelle)

```
GRANT ALL
```

```
ON FornituraNew.*
```

```
TO pippo@localhost
```

```
[IDENTIFIED BY "pippo"];
```

```
[WITH GRANT OPTION];
```

N.B. per i privilegi normali

```
REVOKE ALL
```

```
ON FornituraNew.*
```

```
FROM pippo@localhost;
```

```
GRANT SELECT, INSERT
```

```
ON fornituranew.fornitore
```

```
TO pippo@localhost
```

```
[IDENTIFIED BY "pippo"]
```

```
[WITH GRANT OPTION];
```

N.B. per il privilegio GRANT

```
REVOKE GRANT OPTION
```

```
ON fornituranew.*
```

```
FROM pippo@localhost;
```

N.B. Per controllare i privilegi assegnati ad un determinato utente possiamo utilizzare

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

oppure, nel caso sia quello con il quale siamo loggati,

```
SHOW GRANTS FOR CURRENT_USER;
```

7) usciamo dalla console come DBA

```
quit;  
oppure  
exit;
```

8) eseguiamo il login al db come utente “pippo”

```
mysql -u pippo -p
```

dopo di che verrà richiesta con “Enter password” di digitare la password.

Scrivere “**pippo**” (verranno visualizzati **5 asterischi**)

9) selezioniamo il database come utente “pippo”

```
USE FornituraNew;
```

N.B. Ora l’utente “pippo” può creare le tabelle appartenenti al db “fornituranew”

10) creiamo la tabella “Fornitore”

```
CREATE TABLE Fornitore  
(  
  CodF          VARCHAR(10)      NOT NULL,  
  Cognome       VARCHAR(30)      NOT NULL,  
  Nome          VARCHAR(30)      NOT NULL,  
  DataN         DATE             NOT NULL,  
  PRIMARY KEY (CodF),  
  CHECK (DataN > "1960-01-01")  
) Engine InnoDB CHARACTER SET utf8 COLLATE utf8_bin;
```

(N.B. Il default MYISAM non supporta l’integrità referenziale e le foreign key

(N.B. Il formato data è AAAA-MM-GG o in alternativa AAAA/MM/GG

(N.B. MYSQL non supporta le clausole CHECK...semplicemente le ignora (ev. TRIGGER)
(dalla versione XAMPP 7.4.4 esse invece sono supportate)

11) creiamo la tabella “Articolo”

```
CREATE TABLE Articolo  
(  
  CodA          VARCHAR(10)      NOT NULL,  
  Descrizione   VARCHAR(20)      NOT NULL,  
  Prezzo        DECIMAL (4,2)    NOT NULL,  
  PRIMARY KEY (CodA),  
  CHECK (Prezzo BETWEEN 1.00 AND 10.00)  
) Engine InnoDB CHARACTER SET utf8 COLLATE utf8_bin;
```

N.B. DECIMAL (4,2) significa 4 cifre in totale di cui 2 dopo il punto decimale (analogo a NUMERIC(4,2))

12) creiamo la tabella “Fornisce”

```
(CREATE TABLE Fornisce
(
  IdF          INT(11)          NOT NULL AUTO_INCREMENT,
  CodF1        VARCHAR(10)      NOT NULL,
  CodA1        VARCHAR(10)      NOT NULL,
  DataF        DATE             NOT NULL,
  Qta          INT (1)          NOT NULL,          # DECIMAL (1,0) oppure NUMERIC (1,0)
  PRIMARY KEY (IdF),
  FOREIGN KEY (CodF1) REFERENCES Fornitore (CodF)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
  FOREIGN KEY (CodA1) REFERENCES Articolo (CodA)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
  CHECK (Qta IN (1,2,3))
) Engine InnoDB CHARACTER SET utf8 COLLATE utf8_bin;
```

13) Diamo un’occhiata alle tabelle create finora nel db “fornituranew”

```
SHOW TABLES;
```

N.B. Ora l’utente “pippo” può provvedere a svolgere tutte le query che riterrà opportune

Proviamo a cambiare la struttura della tabella “Fornitore”

14) Aggiungiamo l’attributo non obbligatorio “Email” dichiarato VARCHAR(30)

```
ALTER TABLE Fornitore
  ADD COLUMN Email VARCHAR(30);
```

e guardiamo le modifiche introdotte utilizzando il comando

```
DESCRIBE Fornitore;
```

15) Modifichiamo il tipo dell’attributo “Email” in obbligatorio decimale con 3 cifre per la parte intera e due per la parte decimale

```
ALTER TABLE Fornitore
  MODIFY COLUMN Email DECIMAL (5,2) NOT NULL;
```

e guardiamo le modifiche introdotte utilizzando il comando

```
DESCRIBE Fornitore;
```

16) Oppure modifichiamo il tipo dell’attributo “Email” in opzionale intero di 4 cifre

```
ALTER TABLE Fornitore
  MODIFY COLUMN Email INTEGER (4);
```

e guardiamo le modifiche introdotte utilizzando il comando

```
DESCRIBE Fornitore;
```

N.B Il tipo dati INTEGER (4) equivale a DECIMAL (4,0) o NUMERIC (4,0)

17) Eliminiamo ora l'attributo "Email"

```
ALTER TABLE Fornitore
  DROP COLUMN Email;
```

e guardiamo le modifiche introdotte utilizzando il comando

```
DESCRIBE Fornitore;
```

18) inseriamo nella tabella "Fornitore" le seguenti istanze (n-ple)

- "F_01" (n.b. formato data AAAA/MM/GG o in alternativa AAAA-MM-GG)

```
INSERT INTO Fornitore (CodF, Cognome, Nome, DataN)
  VALUES ('F_01', 'ROSSI', 'MARIO', '1965-01-01');
```

- "F_02" (non è possibile inserire solo qualche valore di attributo da valorizzare essendo tutti obbligatori)

```
INSERT INTO Fornitore
  VALUES ('F_02', 'GIALLI', 'ANDREA', '1960-08-08');
```

- "F_03" (eventualmente è possibile alterare l'ordine delle colonne dando opportunamente i valori)

```
INSERT INTO Fornitore (Cognome, CodF, DataN, Nome,)
  VALUES ('ROSSI', 'F_03', '1963-07-03', 'ELIA');
```

- "F_04"

```
INSERT INSERT INTO Fornitore (CodF, Cognome, Nome, DataN)
  VALUES ('F_04', 'BIANCHI', 'VINCENZO', '1968-11-24');
```

19) inseriamo nella tabella "Articolo" le seguenti istanze (n-ple)

- "A_01"

```
INSERT INTO Articolo (CodA, Descrizione, Prezzo)
  VALUES ('A_01', 'PLUMCAKE', 2.75);
```

- "A_02" (non è possibile inserire solo qualche valore di attributo da valorizzare essendo tutti obbligatori)

```
INSERT INTO Articolo
  VALUES ('A_02', 'CORNFLAKES', 3.50);
```

- "A_03" (eventualmente è possibile alterare l'ordine delle colonne dando opportunamente i valori)

```
INSERT INTO Articolo (Descrizione, CodA, Prezzo)
  VALUES ('ARANCIATA', 'A_03', 1.75);
```

- "A_04"

```
INSERT INTO Articolo  
VALUES ('A_04', 'CARTA IGIENICA', 2.25);
```

- "A_05"

```
INSERT INTO Articolo  
VALUES ('A_05', 'SPAZZOLINO', 1.90);
```

20) inseriamo nella tabella “Fornisce” le seguenti istanze (n-ple)

```
INSERT INTO Fornisce (IdF, CodF1, CodA1, DataF, Qta)  
VALUES (NULL, 'F_01', 'A_01', '2009-10-11', 1);
```

```
INSERT INTO Fornisce  
VALUES (NULL, 'F_01', 'A_02', '2009-11-12', 3);
```

```
INSERT INTO Fornisce  
VALUES (NULL, 'F_02', 'A_01', '2009-12-12', 2);
```

```
INSERT INTO Fornisce  
VALUES (NULL, 'F_02', 'A_03', '2009-12-13', 3);
```

```
INSERT INTO Fornisce  
VALUES (NULL, 'F_03', 'A_04', '2009-07-15', 2);
```

```
INSERT INTO Fornisce  
VALUES (NULL, 'F_03', 'A_03', '2009-07-15', 1);
```

```
INSERT INTO Fornisce  
VALUES (NULL, 'F_03', 'A_02', '2009-08-11', 3);
```

21) aggiorniamo nella tabella “Fornitore” ponendo tutti i cognomi delle istanze ad “ESPOSITO”

```
UPDATE Fornitore  
SET Cognome = "ESPOSITO";
```

N.B. Tutti i record esistenti nella tabella “Fornitore” avranno ora lo stesso cognome (aggiornamento massivo). Solo se la tabella è vuota non ci saranno istanze modificate!

22) aggiorniamo nella tabella “Fornitore” il nome di “ESPOSITO” “ELIA” in “ALEX”

```
UPDATE Fornitore
SET Nome = "ALEX"
WHERE CodF = "F_03";
```

N.B. Verrà aggiornato un solo record nella tabella “Fornitore” (o nessuno se il codice impostato è inesistente)

23) cancelliamo dalla tabella “Fornitore” l’istanza ora aggiornata

```
DELETE FROM Fornitore
WHERE CodF = "F_03";
```

24) cancelliamo dalla tabella “Fornitore” tutte le istanze presenti

```
DELETE FROM Fornitore;
```

N.B. Tutti i record esistenti nella tabella “Fornitore” verranno cancellati (cancellazione massiva). Solo se la tabella è vuota non verrà cancellata alcuna istanza!

25) creiamo la vista (tabella logica non memorizzata fisicamente nel db) che contiene il codice fornitore assieme alla descrizione degli articoli forniti

```
CREATE VIEW Vista1
AS (SELECT CodF, Cognome
    FROM Fornitore);
```

26) aggiorniamo attraverso la vista il cognome del fornitore da “F_01” oppure il suo codice (tale modifica sarà riportata in tutte le tabelle fisiche memorizzate nel db)

```
UPDATE Vista1
SET Cognome = "GENOVESE"
WHERE CodF = "F_01";
oppure
UPDATE Vista1
SET CodF = "X_01"
WHERE CodF = "F_01";
```

27) eliminiamo la vista (tabella logica non memorizzata fisicamente nel db) che contiene il codice

```
DROP VIEW Vista1;
```

28) creiamo la vista (tabella logica non memorizzata fisicamente nel db) che contiene il codice fornitore assieme alla descrizione degli articoli forniti

```
CREATE VIEW Vista2
AS (SELECT CodF, Descrizione
    FROM Fornitore, Fornisce, Articolo
    WHERE (CodF = CodF1) AND (CodA1 = CodA));
```

29) aggiorniamo attraverso la vista la descrizione di tutti gli articoli forniti da “F_01” (tale modifica sarà riportata in tutte le tabelle fisiche memorizzate nel db)

```
UPDATE Vista2  
SET Descrizione = "XXXXXXX"  
WHERE CodF = "F_01";
```

30) eliminiamo la vista (tabella logica non memorizzata fisicamente nel db) che contiene il codice

```
DROP VIEW Vista2;
```

31) creiamo un TRIGGER sulla tabella Articolo che controlli la clausola

CHECK (Prezzo BETWEEN 1.00 AND 10.00)

impostata in precedenza nella CREATE TABLE di Articolo, che MySQL non supporta, inizializzando per esempio a 0 il prezzo di un articolo inserito se esso è inferiore a 1.00 ed a 10.00 se esso è superiore a tale cifra

```
DELIMITER //
CREATE TRIGGER trg1
BEFORE INSERT ON fornituranew.Articolo
FOR EACH ROW
BEGIN
  IF NEW.Prezzo < 1.00
    THEN
      SET NEW.Prezzo = 0;
  END IF;
  IF NEW.Prezzo > 10.00
    THEN
      SET NEW.Prezzo = 10.00;
  END IF;
END //
DELIMITER ;
```

N.B. I delimitatori differenti dallo standard ; vengono utilizzati nell'uso di funzioni, stored procedures, e triggers nelle quali occorre definire istruzioni multiple. E' possibile definire un proprio delimitatore alternativo (nel nostro caso //) per segnalare la fine dell'intera procedura, permettendo nel contempo a ciascuno statement di essere correttamente terminato con il ; Quindi, quando il codice viene eseguito da un client mysql (come la console), il client è in grado di comprendere quando è giunto alla fine della sua definizione ed è in grado di eseguire l'intero set di istruzioni.

Si osservi che il comando DELIMITER è un comando di un client mysql (e di altri tipi di client dbms) e non è uno standard del linguaggio SQL.

N.B. Il qualificatore **NEW** indica proprio che il nome di colonna che stiamo utilizzando si riferisce al nuovo valore della riga che sta per essere aggiornata. NEW si può utilizzare in caso di INSERT e UPDATE. Analogamente è disponibile il qualificatore **OLD** che fa riferimento ai valori precedenti la modifica, e si può utilizzare in caso di UPDATE e DELETE. La modifica attraverso l'istruzione **SET** è possibile solo per i valori NEW e solo nei trigger di tipo BEFORE.

N.B. Attualmente MySQL supporta un solo trigger attivo per ciascuna tabella

32) eliminiamo il TRIGGER precedentemente creato

```
DROP TRIGGER trg1;
```

33) creiamo un TRIGGER sempre sulla tabella Articolo che controlli in modo differente da trg1 la medesima clausola

CHECK (Prezzo BETWEEN 1.00 AND 10.00)

impostata in precedenza nella CREATE TABLE di Articolo, che MySQL non supporta, bloccando l'operazione di inserimento segnalando un opportuno messaggio di testo

```
DELIMITER //
CREATE TRIGGER trg2
BEFORE INSERT ON fornituranew.Articolo
FOR EACH ROW
BEGIN
DECLARE msg VARCHAR(255);
IF NEW.Prezzo < 1.00
THEN
SET msg = concat("Prezzo non puo' essere inferiore a 1.00");
SIGNAL sqlstate '45000' SET message_text = msg;
ELSE
IF NEW.Prezzo > 10.00
THEN
SET msg = concat("Prezzo non puo' essere superiore a 10.00");
SIGNAL sqlstate '45000' SET message_text = msg;
END IF;
END IF;
END //
DELIMITER ;
```

34) eliminiamo il TRIGGER precedentemente creato

```
DROP TRIGGER trg2;
```

35) creiamo un TRIGGER sulla tabella Fornitore che controlli la clausola

CHECK (DataN > "1960-01-01")

impostata in precedenza nella CREATE TABLE di Fornitore, che MySQL non supporta, bloccando l'operazione di inserimento segnalando un opportuno messaggio di testo

```
DELIMITER //
CREATE TRIGGER trg3
BEFORE INSERT ON fornituranew.Fornitore
FOR EACH ROW
BEGIN
DECLARE msg VARCHAR(255) ;
IF NEW.DataN <= "1960-01-01"
THEN
SET msg = concat("DataN non puo' essere precedente al 1 gennaio 1960");
SIGNAL sqlstate '45000' SET message_text = msg;
END IF;
END //
DELIMITER ;
```

36) eliminiamo il TRIGGER precedentemente creato

```
DROP TRIGGER trg3;
```

37) creiamo un altro TRIGGER sulla tabella Fornisce che controlli la clausola

CHECK (Qta IN (1,2,3))

impostata in precedenza nella CREATE TABLE di Fornisce, che MySQL non supporta, bloccando l'operazione di inserimento segnalando un opportuno messaggio di testo

```
DELIMITER //
CREATE TRIGGER trg4
BEFORE INSERT ON fornituranew.Fornisce
FOR EACH ROW
BEGIN
DECLARE msg VARCHAR(255) ;
IF NEW.Qta < 1
THEN
SET msg = concat("Quantita' non puo' essere inferiore a 1");
SIGNAL sqlstate '45000' SET message_text = msg;
ELSE
IF NEW.Qta > 3
THEN
SET msg = concat("Quantita' non puo' essere superiore a 3");
SIGNAL sqlstate '45000' SET message_text = msg;
END IF;
END IF;
END //
DELIMITER ;
```

38) eliminiamo il TRIGGER precedentemente creato

```
DROP TRIGGER trg4;
```

OPERATORI E FUNZIONI SQL

1) Creazione di una tabella con una chiave numerica ad autoincremento

```
CREATE TABLE Tabella
(
id      INT NOT NULL AUTO_INCREMENT ,
nome    VARCHAR( 30 ) NOT NULL ,
prezzo  DECIMAL( 6, 2 ) NOT NULL ,
note    VARCHAR (100),
PRIMARY KEY ( id )
) Engine InnoDB DEFAULT CHARSET=utf8 COLLATE utf8_general_ci;;
```

2) Inserimento di una istanza in una Tabella con chiave numerica ad autoincremento

```
INSERT INTO Tabella (id ,nome ,prezzo, note)
VALUES (NULL , "pippo", 44.25, NULL);
```

oppure nei seguenti modi alternativi

```
INSERT INTO Tabella (id ,nome ,prezzo)
VALUES (NULL , "pippo", 44.25);
```

```
INSERT INTO Tabella (nome ,prezzo)
VALUES ("pippo", 44.25);
```

3) Inserire le seguenti altre due istanze nella Tabella

```
INSERT INTO Tabella
VALUES (NULL , "pluto", 144.75, "praticamente un pazzo");
```

```
INSERT INTO Tabella (id ,nome ,prezzo, note)
VALUES (NULL , "paperino", 200.00, NULL);
```

4) Uso degli operatori IS NULL ed IS NOT NULL

```
SELECT *
FROM Tabella
WHERE note IS NULL;
```

Risultato:

id	nome	prezzo	note
1	pippo	44.25	NULL
3	paperino	200.00	NULL

N.B. se la query viene terminata con \G invece che con il punto e virgola ; i record saranno mostrati verticalmente invece che orizzontalmente

```
SELECT *
FROM Tabella
WHERE note IS NOT NULL;
```

Risultato:

id	nome	prezzo	note
2	pluto	144.75	praticamente un pazzo

5) Uso degli operatori **BETWEEN** ed **NOT BETWEEN**

```
SELECT *
FROM Tabella
WHERE prezzo BETWEEN 50.00 AND 200.00;
```

Risultato:

id	nome	prezzo	note
2	pluto	144.75	praticamente un pazzo
3	paperino	200.00	NULL

```
SELECT *
FROM Tabella
WHERE prezzo NOT BETWEEN 50.00 AND 200.00;
```

Risultato:

id	nome	prezzo	note
1	pippo	44.25	NULL

6) Uso degli operatori **IN** ed **NOT IN**

```
SELECT nome
FROM Tabella
WHERE prezzo IN (44.25, 200.00);
```

Risultato:

nome
pippo
paperino

```
SELECT nome
FROM Tabella
WHERE prezzo NOT IN (44.25, 200.00);
```

Risultato:

nome
pluto

7) Uso degli operatori **LIKE** e **NOT LIKE**

(L'operatore **LIKE** si usa per trovare corrispondenze parziali sulle stringhe attraverso l'impiego dei caratteri jolly "%" che rappresenta "qualsiasi numero di caratteri o nessun carattere", e "_" che invece corrisponde esattamente ad un carattere)

```
SELECT *
FROM Tabella
WHERE nome LIKE "pa%";
```

Risultato:

id	nome	prezzo	note
3	paperino	200.00	NULL

```
SELECT *
FROM Tabella
WHERE nome NOT LIKE "pa%";
```

Risultato:

id	nome	prezzo	note
1	pippo	44.25	NULL
2	pluto	144.75	praticamente un pazzo

```
SELECT *
FROM Tabella
WHERE nome LIKE "p%pp%";
```

Risultato:

id	nome	prezzo	note
1	pippo	44.25	NULL

```
SELECT *
FROM Tabella
WHERE nome LIKE "p__pp%";
```

Risultato:

Empty set (0.00 sec)

```
SELECT *
FROM Tabella
WHERE nome LIKE "p%pp_";
```

Risultato:

id	nome	prezzo	note
1	pippo	44.25	NULL

```
SELECT *
FROM Tabella
WHERE nome LIKE "_____";
```

Risultato:

id	nome	prezzo	note
1	pippo	44.25	NULL
2	pluto	144.75	praticamente un pazzo

8) Funzioni sulle stringhe **CONCAT** e **CONCAT_WS**

(si utilizzano per concatenare due o più stringhe, nel secondo caso aggiungendo un separatore)

```
SELECT CONCAT_WS ('+',id, nome, note) AS Concatenazione
FROM Tabella;
```

Risultato(N.B. la funzione CONCAT_WS concatena anche le n-ple che hanno valore NULL) :

Concatenazione
1+pippo
2+pluto+praticamente un pazzo
3+paperino

```
SELECT CONCAT (id,'+', nome,'+', note) AS Concatenazione
FROM Tabella;
```

Risultato (N.B. la funzione CONCAT da NULL se uno dei campi considerati è NULL)

Concatenazione
NULL
2+pluto+praticamente un pazzo
NULL

9) Funzioni sulle stringhe **LOWER** ed **UPPER**

(consentono di trasformare una stringa, rispettivamente, in tutta minuscola o tutta maiuscola)

```
SELECT LOWER (note) AS Tutto_Minuscolo
FROM Tabella;
```

Risultato:

Tutto_Minuscolo
NULL
praticamente un pazzo
NULL

```
SELECT UPPER (note) AS 'Tutto Maiuscolo'
FROM Tabella;
```

Risultato:

Tutto_Maiuscolo
NULL
PRATICAMENTE UN PAZZO
NULL

10) Funzioni sulle stringhe **LEFT** e **RIGHT**

(estraggono n caratteri a sinistra o a destra della stringa)

```
SELECT LEFT (nome, 2) AS Sinistra
FROM Tabella;
```

Risultato:

Sinistra
pi
pl
pa

```
SELECT RIGHT (nome, 3) AS Destra
FROM Tabella;
```

Risultato:

Destra
ppo
uto
ino

11) Funzioni sulle stringhe **LENGTH** e **CHAR_LENGTH**

(restituiscono la lunghezza di una stringa, con la differenza che la prima misura la lunghezza in byte, mentre la seconda restituisce il numero di caratteri; evidentemente i valori saranno diversi per le stringhe che contengono caratteri multi-byte)

```
SELECT nome, LENGTH (nome) AS Lunghezza
FROM Tabella;
```

Risultato:

nome	Lunghezza
pippo	5
pluto	5
paperino	8

Risultato medesimo in questo caso (nessun carattere multibyte):

nome	Lunghezza
pippo	5
pluto	5
paperino	8

12) Funzioni sulle stringhe **LPAD** e **RPAD**

(aggiungono, a sinistra (LPAD) o a destra, i caratteri necessari a portare la stringa alla lunghezza specificata, eventualmente accorciandola se più lunga)

```
SELECT LPAD (nome, 7, '_')
FROM Tabella;
```

Risultato:

LPAD (nome, 7, '_')
_pippo
_pluto
paperin

```
SELECT RPAD (nome, 7, '*')
FROM Tabella;
```

Risultato:

RPAD (nome, 7, '*')
pippo**
pluto**
paperin

13) Funzioni sulle stringhe **LTRIM** e **RTRIM**

(eliminano gli spazi a sinistra (LTRIM) o a destra (RTRIM) di una stringa purchè diversa da NULL)

```
SELECT LTRIM ( RIGHT (note, 6))
FROM Tabella;
```

Risultato:

LTRIM (RIGHT (note, 6))
NULL
pazzo
NULL

N.B. Si noti l'effetto di LTRIM guardando l'output della sola istruzione RIGHT (note, 6) in cui c'è lo spazio iniziale:

```
SELECT RIGHT (note, 6)
FROM Tabella;
```

Risultato:

RIGHT (note, 6)
NULL
pazzo
NULL

14) Funzioni sulle stringhe **SUBSTRING**

(restituisce una parte della stringa, a partire dal carattere specificato fino alla fine della stringa o, se indicato, per un certo numero di caratteri)

```
SELECT SUBSTRING ('accidenti', 5);
```

Risultato:

substring('accidenti',5)
denti

```
SELECT SUBSTRING ('accidenti', 5, 3);
```

Risultato:

substring('accidenti',5,3)
den

15) Funzioni sulle stringhe **REVERSE** e **REPLACE**

(la prima inverte una stringa mentre la seconda sostituisce in una stringa una sequenza di caratteri in un'altra)

```
SELECT REVERSE ('accidenti');
```

Risultato:

REVERSE ('accidenti')
itnedicca

```
SELECT REPLACE ('uccello','cc','g');
```

Risultato:

REPLACE ('uccello','cc','g')
ugello

16) Funzione matematica **ABS**

(restituisce il valore assoluto (non segnato) di un numero)

```
SELECT ABS (-34.65) AS 'Valore assoluto';
```

Risultato:

Valore assoluto
34.65

17) Funzione matematica **POWER**

(effettua l'elevamento a potenza (richiede base ed esponente))

```
SELECT POWER (2,8) AS Potenza;
```

Risultato:

Potenza
256

18) Funzione matematica **RAND**

(genera un valore casuale compreso tra 0 e 1)

```
SELECT id, RAND () AS Random
FROM Tabella;
```

Risultato:

id	Random
1	0.4945261743800027
2	0.2950904753944748
3	0.9918738845660108

```
SELECT RAND () AS Random;
```

Risultato:

Random
0.07409802738027463

19) Funzioni matematiche di arrotondamento **FLOOR**, **CEILING**, **ROUND** e **TRUNCATE**

- **FLOOR** (arrotonda all'intero inferiore)
- **CEILING** (all'intero superiore)
- **ROUND** (arrotonda all'intero superiore da .5 in su, altrimenti all'inferiore)
- **TRUNCATE** che tronca il numero (non arrotonda) alla quantità specificata di decimali

```
SELECT FLOOR (prezzo) AS 'Uso Floor'
FROM Tabella;
```

Risultato:

Uso Floor
44
144
200

```
SELECT CEILING (prezzo) AS 'Uso Ceiling'
FROM Tabella;
```

Risultato:

Uso Ceiling
45
145
200

```
SELECT ROUND (prezzo) AS 'Uso Round'
FROM Tabella;
```

Risultato:

Uso Round
44
145
200

```
SELECT TRUNCATE (prezzo, 1) AS 'Uso Truncate'
FROM Tabella;
```

Risultato:

Uso Truncate
44.2
144.7
200.0

N.B. Se abbiamo bisogno di generare un intero compreso fra x e y, possiamo usare questa formula: $\text{FLOOR}(x + \text{RAND}() * (y - x + 1))$. Supponiamo di voler generare un numero casuale tra 1 e 100

```
SELECT FLOOR (1 + RAND () * 100) AS 'Numero casuale';
```

Risultato:

Numero casuale
53

20) Funzioni sulle date

- **CURRENT_DATE** restituisce la data attuale (formato YYYY-MM-DD)
- **CURRENT_TIME** restituisce l'ora attuale (formato HH-MM-SS);
- **CURRENT_TIMESTAMP** e **NOW**, che sono sinonimi, restituiscono data e ora.
- **DATE** estrae la parte data da un timestamp;
- **TIME** fa lo stesso con la parte ora

```
SELECT CURRENT_DATE ();
```

Risultato:

CURRENT_DATE ()
2015-03-26

```
SELECT FROM_UNIXTIME (UNIX_TIMESTAMP ( ), '%d-%m-%Y');
```

Risultato:

FROM_UNIXTIME (UNIX_TIMESTAMP (), '%d-%m-%Y')
26-03-2015

```
SELECT FROM_UNIXTIME (UNIX_TIMESTAMP ( ), '%d-%m-%Y %H.%i.%s');
```

Risultato:

FROM_UNIXTIME (UNIX_TIMESTAMP (), '%d-%m-%Y %H.%i.%s')
26-03-2015 11.50.55

```
SELECT CURRENT_TIME ();
```

Risultato:

CURRENT_TIME ()
11:54:09

```
SELECT NOW ();
```

Risultato:

NOW()
2015-03-26 11:56:56

Ci sono poi, come anticipato, le funzioni che estraggono da una data, da un orario o da un timestamp le singole informazioni:

- **DAY** restituisce il giorno del mese,
- **DAYOFWEEK** restituisce il giorno della settimana (1 è la domenica e 7 il sabato),
- **DAYOFYEAR** restituisce il giorno dell'anno (da 1 a 366),
- **MONTH** restituisce il mese,
- **YEAR** restituisce l'anno;
- **hour** restituisce l'ora,
- **MINUTE** restituisce i minuti,
- **SECOND** restituisce i secondi

```
SELECT DATE (NOW());
```

N.B. (equivale a `SELECT CURRENT_DATE()`)

Risultato:

DATE (NOW())
2015-03-26

```
SELECT TIME (NOW());
```

N.B. (equivale a `SELECT CURRENT_TIME()`)

Risultato:

TIME (NOW())
15:30:19

```
SELECT DAY (NOW());
```

Risultato:

DAY(NOW())
26

```
SELECT DAYOFWEEK (NOW());
```

// dom = 0, lun = 1, mar = 2, merc = 3, etc, etc

Risultato:

DAYOFWEEK(NOW())
5

```
SELECT DAYOFYEAR (NOW());
```

// 1 gennaio = giorno 1, etc. etc.

Risultato:

DAYOFYEAR(NOW())
85

```
SELECT MONTH (NOW());
```

// genn = 1, febb = 2, mar = 3, etc. etc.

Risultato:

MONTH(NOW())
3

```
SELECT YEAR (NOW());
```

Risultato:

YEAR(NOW())
2015

```
SELECT HOUR(NOW()) AS ORA, MINUTE(NOW()) AS MINUTI, SECOND(NOW()) AS SECONDI;
```

Risultato:

ORA	MINUTI	SECONDI
15	44	49

Ci sono infine due funzioni (**DATE_FORMAT** e **TIME_FORMAT**) che permettono di formattare i valori di date e ore. Esse richiedono in input una data o un timestamp (**DATE_FORMAT**) o un orario (**TIME_FORMAT**), nonché una stringa contenente i simboli per la formattazione voluta

```
SELECT DATE_FORMAT (NOW(), '%d/%m/%Y');
```

Risultato:

DATE_FORMAT (NOW(), '%d/%m/%Y')
26/03/2015

```
SELECT TIME_FORMAT (NOW(), '%H:%i:%s');
```

Risultato:

TIME_FORMAT (NOW(), '%H:%i:%s')
15:53:19

Elenchiamo alcuni di questi simboli seguiti da qualche esempio:

- Anno: '%Y' (4 cifre), '%y' (2 cifre)
- Mese: '%m' (2 cifre), '%c' (1 o 2 cifre), '%M' (nome intero in inglese), '%b' (nome abbreviato)
- Giorno del mese: '%d' (due cifre), '%e' (1 o 2 cifre)
- Giorno della settimana: '%W' (nome intero in inglese), '%a' (nome abbreviato), '%w' (numerico con 0 = domenica e 6 = sabato)
- Ore: '%H' (2 cifre), '%k' (1 o 2 cifre), '%h' (2 cifre da 01 a 12), '%l' (1 o 2 cifre da 1 a 12)
- Minuti: '%i' (2 cifre)
- Secondi: '%s' (2 cifre)

```
SELECT DATE_FORMAT (NOW(), '%d/%M/%Y');
```

Risultato:

DATE_FORMAT (NOW(), '%d/%M/%Y')
26/March/15

```
SELECT TIME_FORMAT (NOW(), '%l:%i:%s');
```

Risultato:

TIME_FORMAT (NOW(), '%l:%i:%s')
3:54:59

21) Funzioni condizionali in MySQL: CASE, IF, IFNULL e NULLIF

CASE

(Mediante il costrutto CASE ... WHEN è possibile creare delle istruzioni condizionali piuttosto complesse prevedendo non solo un'unica condizione, ma una serie di condizioni alternative)

```
SELECT CASE id
      WHEN 1 THEN 'primo'
      WHEN 2 THEN 'secondo'
      ELSE 'altro'
      END
FROM Tabella;
```

Risultato:

primo
secondo
altro

IF

(La funzione IF è più semplice di CASE e consente di verificare un'unica condizione. La funzione prevede tre argomenti: il primo è l'espressione da verificare, il secondo è il risultato restituito in caso in cui l'espressione sia vera, il terzo il risultato da restituire nel caso l'espressione sia falsa)

```
SELECT id, nome, prezzo,
      IF ((prezzo > 100.00 AND prezzo < 200.00), 'Medio Alto', 'Scarto') AS valutazione
FROM Tabella;
```

Risultato:

id	nome	prezzo	valutazione
1	pippo	44.25	Scarto
2	pluto	144.75	Medio Alto
3	paperino	200.00	Scarto

IFNULL

(La funzione IFNULL intercetta eventuali errori del MySQL (che restituisce NULL) e li gestisce in modo predeterminato. La funzione ISNULL accetta due parametri: il primo è il dato da verificare, il secondo è il risultato da restituire qualora il primo argomento sia NULL)

```
SELECT id, nome, note, IFNULL(note, 'Questo campo è a NULL') AS 'Esito IF'
FROM Tabella;
```

Risultato:

id	nome	note	Esito IF
1	pippo	NULL	Questo campo è a NULL
2	pluto	praticamente un pazzo	praticamente un pazzo
3	paperino	NULL	Questo campo è a NULL

NULLIF

(Questa funzione accetta due argomenti che vengono confrontati tra loro; se sono uguali restituisce NULL in caso contrario restituisce il primo argomento)

```
SELECT *
INTO newtab
FROM Tabella AS t1, Tabella AS t2;
```

Risultato:

Query parametriche (da linea di comando)

Innanzitutto in modalità MySql console occorre definire il parametro ed assegnargli un valore

```
set @NomeX = "MARIO";
```

per poi eseguire la query in oggetto

```
SELECT *
INTO Fornitore
FROM Nome = @NomeX;
```

che restituirà

CodF	Cognome	Nome	DataN
F_01	ROSSI	MARIO	1965-01-01

Query da file

Creiamo, prima di ogni cosa, il file "insert_articolo.sql" contenente le istruzioni da eseguire sul db (ad esempio tre insert sulla Tabella Articolo) all'interno della directory che ospita l'eseguibile di mysql (directory *usr/local/mysql/bin*)

```
INSERT INTO Articolo
VALUES ("X_01", "LATTE", 11.25);

INSERT INTO Articolo
VALUES ("X_02", "POMATA", 17.25);

INSERT INTO Articolo
VALUES ("X_03", "VINO", 10.75);
```

per poi eseguire la query in oggetto da linea di comando tramite il comando "source"

```
source insert_articolo.sql;
```

Verranno aggiunte le tre nuove n-ple nella Tabella in oggetto.