

SQL

- *Linguaggio Standard*
- *Estensioni di MySQL*
- *Interfacciamento con Java e PHP*
- *Elementi di Sicurezza*

Giuseppe Della Penna
Università degli Studi di L'Aquila

giuseppe.dellapenna@univaq.it
<http://people.disim.univaq.it/dellapenna>

Versione documento: 230524



This work is licensed under CC BY-NC-SA 4.0. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/4.0/>

SQL: Elementi di Base

Tipi di dato, operatori e funzioni di sistema





Tipi di dato

- › I tipi di dato in SQL:1999 si suddividono in:
 - tipi *predefiniti*
 - tipi *strutturati*
 - tipi *user-defined*.
- › I tipi di dato predefiniti sono ulteriormente suddivisi in 5 categorie: tipi numerici, tipi binari, tipi carattere, tipi temporali, tipi booleani.



Tipi numerici esatti

- › Rappresentano valori interi e valori decimali in virgola fissa (es. 75, 4.5, -6.2)

INTEGER	Rappresenta i valori interi. La precisione (numero totale di cifre) di questo tipo di dato è espressa in numero di bit o cifre, a seconda della specifica implementazione di SQL.
SMALLINT	Rappresenta i valori interi. I valori di questo tipo sono usati per eventuali ottimizzazioni poichè richiedono minore spazio di memorizzazione. L'unico requisito è che la precisione di questo tipo di dato sia non maggiore della precisione del tipo di dato INTEGER.
NUMERIC(p,s)	Rappresenta i valori decimali. È caratterizzato da una precisione e una scala (rispettivamente numero totale di cifre da rappresentare e numero di cifre della parte frazionaria).
DECIMAL(p,s)	È simile al tipo NUMERIC. La differenza tra NUMERIC e DECIMAL è che il primo deve essere implementato esattamente con la precisione richiesta, mentre il secondo può avere una precisione maggiore (in pratica, il numero viene rappresentato internamente con una precisione possibilmente maggiore e approssimato alla precisione data solo per la visualizzazione).



Tipi numerici approssimati

- › Rappresentano valori reali in virgola mobile (es. $1256e^{-4}$).

REAL	Rappresenta valori a singola precisione in virgola mobile. La precisione dipende dalla specifica implementazione di SQL.
DOUBLE PRECISION	Rappresenta valori a doppia precisione in virgola mobile. La precisione dipende dalla specifica implementazione di SQL.
FLOAT(p)	Permette di richiedere la precisione che si desidera (il parametro p indica il numero di cifre da rappresentare).



Tipi binari

- › Questi tipi sono principalmente usati per inserire nella base di dati contenuto non testuale, ad esempio immagini o generici files. La manipolazione di questo tipo di dati non è sempre possibile tramite l'SQL interattivo.
- › Inserire binari nelle basi di dati presenta una serie di pericoli riguardanti le performance e la sicurezza che vanno sempre tenuti in considerazione!

BIT(n)	Rappresenta stringhe di bit di lunghezza fissata, dove n è la lunghezza massima della stringa (se non viene specificata alcuna lunghezza, il default è 1).
BIT VARYING(n)	Rappresenta stringhe di bit a lunghezza variabile con lunghezza massima predefinita, dove n è la lunghezza massima delle stringhe. La differenza con il tipo BIT è che con questo tipo lo spazio allocato per ogni stringa corrisponde sempre alla lunghezza massima predefinita, mentre per BIT VARYING si usano strategie diverse, allocando uno spazio proporzionale alla effettiva lunghezza della stringa inserita. Per specificare i valori di entrambi i tipi possono essere utilizzate le rappresentazioni binaria o esadecimale (es. 01111 o 0x44).
BLOB	Questo tipo di dato permette di definire sequenze di bit di elevate dimensioni (tipicamente grossi files). Rispetto ai tipi BIT, un BLOB può arrivare alle dimensioni di megabyte o gigabyte. I dati associati vengono memorizzati in files separati, caricati solo se richiesti, e non inclusi nei record, in modo da rendere più rapido l'accesso alle tabelle.



Tipi carattere

- › Sono tipi usati per rappresentare caratteri, stringhe o interi blocchi di testo. È possibile associare ai tipi testo un CHARACTER SET di riferimento e la relativa COLLATION (ordine dei caratteri nel set).

CHARACTER(n) CHAR(n)	Questo tipo di dato permette di definire stringhe di caratteri di lunghezza fissata, dove n è la lunghezza massima delle stringhe (se non viene specificata alcuna lunghezza, il default è 1).
CHARACTER VARYING(n) VARCHAR(n)	Questo tipo di dato permette di definire stringhe di caratteri a lunghezza variabile con lunghezza massima predefinita, dove n è la lunghezza massima delle stringhe. La differenza con il tipo CHAR è la stessa descritta per i tipi BIT e BIT VARYING.
CLOB	Questo tipo di dato permette di definire sequenze di caratteri di elevate dimensioni (blocchi di testo), similmente a quello che accede con i BLOB.



Tipi temporali

- › Permettono di gestire date e ore. Tipicamente il formato usato per specificare queste due informazioni è quello anglosassone, quindi una data dovrà essere inserita come 'anno-mese-giorno' e un'ora come 'ore:minuti:secondi'. Inoltre, nella maggior parte dei DBMS date e ore sono inserite sotto forma di stringhe di caratteri, per cui vanno racchiuse tra apici.

DATE	Rappresenta le date espresse come anno (4 cifre), mese (2 cifre comprese tra 1 e 12), giorno (2 cifre comprese tra 1 e 31 con ulteriori restrizioni a seconda del mese).
TIME	Rappresenta i tempi espressi come ora (2 cifre), minuti (2 cifre) e secondi (2 cifre).
TIMESTAMP	Rappresenta una "concatenazione" dei due tipi di dato precedenti con una precisione al microsecondo, pertanto permette di rappresentare valori temporali che consistono di anno, mese, giorno, ora, minuti, secondi e microsecondi (di solito specificati nella forma 'anno-mese-giorno ore:minuti:secondi.microsecondi').



Tipi booleani

- › I tipi booleani non sono presenti in tutti i DBMS. Spesso sono sostituiti con un dominio definito dall'utente o con numeri (0/1) o caratteri (y/n)

BOOLEAN Rappresenta valori booleani. I valori di tale tipo sono TRUE, FALSE, UNKNOWN.

- › La logica a tre valori dell'SQL, che include anche il valore UNKNOWN, è utilizzata per trattare i casi in cui alcuni dati presenti in operazioni quali i confronti non siano disponibili (nulli). Le tabelle di verità delle tre operazioni fondamentali, estese con il valore UNKNOWN (?), sono le seguenti.

AND			
	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR			
	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?



Valori null

- › Il valore null è una speciale costante presente in tutti i DBMS, e rappresenta "assenza di informazione". E' compatibile con ogni tipo di dato e si usa per dare valore ad attributi non applicabili o dall'effettivo valore non noto.



Tipi di dato in MySQL

- › MySQL conserva i principali tipi di dato SQL e aggiunge alcune estensioni, definendo range più precisi per lunghezze e precisioni.
- › Esistono anche altri tipi oltre a quelli elencati qui, che sono i più comuni.
- › I tipi interi hanno tutti la variante **UNSIGNED**, con lo stesso numero di bytes ma a partire da zero (quindi, ad esempio, invece di -128 – 127 il range di **UNSIGNED TINYINT** è 0-255)

CHAR	lunghezza massima 255
VARCHAR	lunghezza massima 65535
TINYTEXT	Come CLOB - lunghezza massima 255
TEXT	Come CLOB - lunghezza massima 65535
BLOB	Come BLOB - lunghezza massima 65535
MEDIUMTEXT	Come CLOB - lunghezza massima 16000
MEDIUMBLOB	Come BLOB - lunghezza massima 16000
LONGTEXT	Come CLOB - lunghezza massima 4Gb
LONGBLOB	Come BLOB - lunghezza massima 4Gb
TINYINT	Come SMALLINT (-128 - 127)
SMALLINT	Come SMALLINT (-32768 - 32767)
MEDIUMINT	Come INTEGER (-8388608 - 8388607)
INT	Come INTEGER (-2147483648 - 2147483647)
BIGINT	Come INTEGER (-9223372036854775808 - 9223372036854775807)
FLOAT	-3.402823466E+38 - -1.175494351E-38, 0, 1.175494351E-38 - 3.402823466E+38. Precisione di default 23 digit
DOUBLE	-1.7976931348623157E+308 - -2.2250738585072014E-308, 0, 2250738585072014E-308 - 1.7976931348623157E+308
DECIMAL	Massimo 65 digit
DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS (anni dal 1000 a 9999)
TIMESTAMP	Simile a DATETIME (anni dal 1970 al 2038, compatibili con gli Unix timestamp)
TIME	HH:MM:SS
ENUM('a','b')	Stringa con esattamente uno dei valori elencati
SET('a','b')	Stringa con uno o più dei valori elencati, separati con virgole



Domini

- › SQL permette di definire dei domini e utilizzarli nella definizione di tabelle. Un dominio è un tipo di dato derivato, basato su uno dei tipi semplici visti finora, cui si possono associare particolari vincoli o informazioni.
- › La definizione dei domini fa propriamente parte del DDL.
- › Non sono supportati in MySQL.



Domini

Creazione

- › **CREATE DOMAIN *d* AS *tipo* [DEFAULT *valore*] [CHECK(*vincolo*)]**
definisce il dominio *d* (il nome deve essere una stringa alfanumerica), basandolo sul *tipo* dato. Opzionalmente è possibile specificare:
 - un *valore* di default per i valori associati a questo dominio (il default verrà usato per inizializzare tutti gli attributi dichiarati usando il dominio come tipo)
 - un *vincolo* di tipo CHECK. Un vincolo check è definito come un predicato sullo speciale argomento VALUE, che rappresenta i possibili valori ammessi nel dominio, composto usando tutte le operazioni comuni dell'SQL. Il dominio conterrà i soli valori per cui la valutazione del predicato risulta vera.



Esempi

```
CREATE DOMAIN tipoEta  
AS INTEGER  
CHECK(VALUE BETWEEN 1 AND 120);
```

```
CREATE DOMAIN occupazione  
AS VARCHAR(30)  
DEFAULT 'disoccupato';
```

```
CREATE DOMAIN DominioMansione  
AS CHAR(10)  
CHECK (VALUE IN('DIRIGENTE','INGEGNERE','TECNICO','SEGRETARIA'));
```



Domini

Modifica

- › **ALTER DOMAIN *d* SET DEFAULT *valore***
modifica il valore di default del dominio *d* in *valore*.
- › **ALTER DOMAIN *d* DROP DEFAULT**
elimina il valore di default del dominio *d*.
- › **ALTER DOMAIN *d* DROP CONSTRAINT**
rimuove il vincolo (check) dal dominio *d*.
- › **ALTER DOMAIN *d* ADD CONSTRAINT *CHECK(vincolo)***
aggiunge il *vincolo* check specificato al dominio *d*.



Domini

Cancellazione

› **DROP DOMAIN d RESTRICT | CASCADE**

rimuove il dominio d

- se viene specificato RESTRICT: il dominio viene rimosso solo se nessuna tabella lo utilizza
- se viene specificato CASCADE: in ogni tabella che lo utilizza il nome del dominio viene sostituito dalla sua definizione (la modifica non influenza i dati presenti nella tabella)



Operatori e funzioni comuni dell'SQL

- › Il linguaggio SQL definisce una serie di operatori e funzioni di base che possono essere usati in diversi comandi, ad esempio per creare vincoli CHECK, per filtrare i record di una o più tabelle, o per derivare valori dai dati presenti nel database.



Funzioni aritmetiche di base

- › SQL mette a disposizione tutte le usuali funzioni aritmetiche, indicate con gli operatori $+$, $-$, $*$ e $/$.
- › Nel calcolo delle espressioni aritmetiche, il valore nullo viene considerato uguale al valore zero.



Funzioni scalari

- › Sebbene non siano fornite in tutte le implementazioni di SQL, le funzioni che seguono sono molto comuni.
 - **ABS(n)** calcola il valore assoluto del valore numerico n.
 - **MOD(n,b)** calcola il resto intero della divisione di n per b.
 - **SQRT(n)** calcola la radice quadrata di n.
- › Alcuni DBMS supportano anche funzioni trigonometriche e funzioni per il calcolo della parte intera superiore ed inferiore.



Funzioni scalari in MYSQL

› MySQL dispone di un set esteso di funzioni scalari:

- ABS
- COS
- SIGN
- SIN
- MOD
- TAN
- FLOOR
- ACOS
- CEILING
- ASIN
- ROUND
- ATAN, ATAN2
- DIV
- COT
- EXP
- RAND
- LN
- LEAST
- LOG, LOG2, LOG10
- GREATEST
- POW
- DEGREES
- POWER
- RADIANS
- SQRT
- TRUNCATE
- PI



Espressioni e Funzioni per Stringhe

- › Questi operatori si applicano ai tipi carattere.
 - **LENGTH(s)** calcola la lunghezza della stringa s.
 - **SUBSTR(s, m, [n])** (m ed n sono interi) estrae dalla stringa s la sottostringa dal carattere di posizione m per una lunghezza n (se n è specificato) oppure fino all'ultimo carattere.
 - **s1 || s2** restituisce la concatenazione delle due stringhe s1 e s2.
- › Anche in questo caso, diversi dialetti di SQL potrebbero offrire insieme di funzioni diversi.



Espressioni e Funzioni per Stringhe in MYSQL

› MySQL dispone di un set esteso di funzioni per le stringhe:

- ASCII
- SUBSTRING
- ORD
- MID
- CONV
- SUBSTRING_INDEX
- BIN
- LTRIM
- OCT
- RTRIM
- HEX
- TRIM
- CHAR
- SOUNDEX
- CONCAT
- SPACE
- CONCAT_WS
- REPLACE
- LENGTH
- REPEAT
- CHAR_LENGTH
- REVERSE
- BIT_LENGTH
- INSERT
- LOCATE
- ELT
- INSTR
- FIELD
- LPAD
- LCASE
- RPAD
- UCASE
- LEFT
- LOAD_FILE
- RIGHT
- QUOTE



Funzioni e costanti per tipi temporali

- › Le funzioni temporali sono quelle più variabili tra DBMS diversi. Le costanti temporali sono invece di solito ampiamente supportate
 - **DATE(v), TIME(v), TIMESTAMP(v)** convertono rispettivamente un valore scalare in una data, un tempo, un timestamp.
 - **CHAR(d, [f])** converte un valore di data/tempo d in una stringa di caratteri; può inoltre ricevere una specifica di formato f.
 - **DAYS(v)** converte una data v in un intero che rappresenta il numero di giorni a partire dall'anno zero.
 - **CURRENT_DATE**: rappresenta la data corrente
 - **CURRENT_TIME**: rappresenta l'ora corrente
 - **CURRENT_TIMESTAMP**: rappresenta il timestamp corrente
- › Date e tempi possono anche essere usati in espressioni aritmetiche: in tali espressioni è possibile usare diverse unità temporali quali **YEAR[S], MONTH[S], DAY[S], HOUR[S], MINUTE[S], SECOND[S]**, da posporre ai numeri.



Funzioni e costanti per tipi temporali in MYSQL

› MySQL dispone di un set esteso di funzioni per la manipolazione di date:

- DAYOFWEEK
- DATE_SUB
- WEEKDAY
- ADDDATE
- DAYOFMONTH
- SUBDATE
- DAYOFYEAR
- EXTRACT
- MONTH
- TO_DAYS
- DAYNAME
- FROM_DAYS
- MONTHNAME
- DATE_FORMAT
- QUARTER
- TIME_FORMAT
- WEEK
- CURRENT_DATE
- YEAR
- CURRENT_TIME
- YEARWEEK
- NOW
- HOUR
- SYSDATE
- MINUTE
- UNIX_TIMESTAMP
- SECOND
- FROM_UNIXTIME
- PERIOD_ADD
- SEC_TO_TIME
- PERIOD_DIFF
- TIME_TO_SEC
- DATE_ADD



Esempi

//per provare queste espressioni in un DBMS, occorre come vedremo
selezionarne il valore, scrivendole all'interno di una clausola SELECT

```
SELECT TO_DAYS('2001-8-1')  
//731063
```

```
DATE('1997-6-20')  
//1997-06-20
```

```
SELECT DATE_FORMAT(data_assunzione, '%d-%b-%Y')  
//la data di assunzione in formato giorno-iniziali mese-anno
```

```
SELECT DATE_ADD('1997-6-20', INTERVAL 90 DAY)  
//1997-09-18
```

```
SELECT (DATE_ADD(data, INTERVAL 90 DAY) >= CURRENT_DATE)  
//l'espressione è vera se la data è al più tre mesi nel passato.
```



Operatori di confronto

- › SQL mette a disposizione tutti gli usuali operatori di confronto, cioè maggiore >, minore <, maggiore o uguale >=, minore o uguale <=, uguale =, diverso <>.
- › Nella valutazione di questi operatori, la presenza di almeno un operando nullo porta alla generazione di un valore logico UNKNOWN come risultato.
- › E' inoltre presente un operatore "scorciatoia" per controllare se un valore rientra in un intervallo:
 - ***c* BETWEEN *v1* AND *v2***
c* NOT BETWEEN *v1* AND *v2
le due espressioni sono vere se, rispettivamente, il valore *c* è compreso o non compreso tra i valori *v1* e *v2*



Ricerca di valori in un insieme

- › L'operatore IN permette di determinare se un valore si trova in un insieme specificato.
- › **c IN ($v1, v2, \dots, vn$)**
 c NOT IN ($v1, v2, \dots, vn$)
le espressioni sono vere se, rispettivamente, il valore c è compreso o non è compreso tra i valori $v1, v2, \dots, vn$.
- › **c IN *query***
 c NOT IN *query*
in questo caso la lista di valori non è dichiarata esplicitamente, ma viene generata da una interrogazione (*query*) sulla base di dati. Questa interrogazione deve necessariamente restituire una tabella costituita da una sola colonna, come vedremo più avanti.



Confronto tra stringhe di caratteri

- › L'operatore LIKE permette di eseguire alcune semplici operazioni di *pattern matching* su valori di tipo stringa.
 - ***c* LIKE *pattern***
l'espressione è vera se il *pattern* fa match con la stringa *c*.
- › Un pattern è una stringa di caratteri che può contenere i caratteri speciali (*metacaratteri* o *wildcards*) % e _
 - il metacarattere % denota una sequenza di caratteri arbitrari di lunghezza qualsiasi (anche zero);
 - il metacarattere _ denota esattamente un carattere
 - ogni altro carattere nel pattern fa match solo con un carattere identico nella stringa.



Esempi

//qui proviamo l'operatore in una clausola SELECT, ma può essere usato ovunque sia richiesto un valore booleano!

```
SELECT 'MINESTRA' LIKE 'M%A' //1 (vero)
SELECT 'MAMMA' LIKE 'M%A' //1 (vero)
SELECT 'MA' LIKE 'M%A' //1 (vero)
```

```
SELECT 'MINESTRA' LIKE '%MA' //0 (falso)
SELECT 'MAMMA' LIKE '%MA' //1 (vero)
SELECT 'MA' LIKE '%MA' //1 (vero)
SELECT 'PROGRAMMA' LIKE '%MA' //1 (vero)
```

```
SELECT nome LIKE '%MA%' //vero se la stringa nome contiene la sottostringa 'MA'
```

```
SELECT nome LIKE '___' //vero se il nome è di tre caratteri
```

```
SELECT 'ABRACADABRA' LIKE 'A_R%' //1 (vero)
SELECT 'ARRIVO' LIKE 'A_R%' //1 (vero)
```



Operatori logici

- › SQL mette a disposizione i quattro usuali operatori logici, indicati con le parole chiave AND, OR e NOT. Questi operatori possono essere usati per costruire condizioni logiche complesse a partire dai test semplici visti finora.



Trattamento dei valori null

- › In SQL-89, la presenza di valori null rende ogni predicato falso.
- › Nelle versioni successive di SQL, si può fare uso del predicato **IS [NOT] NULL** per effettuare un test sulla presenza o assenza di valori nulli.
- › Per trattare i valori null, nel caso l'utente non specifichi esplicitamente un predicato IS [NOT] NULL, SQL usa la logica a tre valori vista nel capitolo tipi di dato.

SQL: DDL

Creazione, modifica ed eliminazione dei metadati





Gestione degli Utenti

› I DBMS hanno una gestione degli utenti molto raffinata, che permette di assegnare specifici privilegi di accesso a interi database e a tabelle.

› Per prima cosa, è necessario creare gli utenti:

CREATE USER *u* IDENTIFIED BY *p*

Crea l'utente *u* e gli assegna la password *p*

– In MySQL, *u* e *p* sono stringhe e andrebbero poste tra virgolette.

– In MySQL, i nomi di utente devono avere la forma 'utente'@'host', dove host è la macchina dalla quale l'utente può collegarsi al database. Se non lo si specifica (scrivendo 'utente' o 'utente@'%) tutte le macchine saranno accettabili.

Solitamente, è meglio restringere l'accesso agli utenti locali (@'localhost')

› **DROP USER *n***

elimina l'utente specificato

› **ALTER USER *n* IDENTIFIED BY *p***

cambia la password per l'utente specificato



Creazione di Database

- › Un DBMS può contenere tabelle appartenenti a diversi schemi relazionali. Per distinguerle, queste vengono poste in *namespace* separati.
- › I namespace sono chiamati dallo standard SQL *schema*, ma spesso è presente anche il più intuitivo sinonimo *database*.
- › Quando, in un'istruzione SQL, si usa un nome di tabella, lo si può far precedere da un prefisso indicante lo schema a cui appartiene, scrivendo *schema.tabella*.
 - Se non si specifica un nome di schema, e la tabella è presente in schemi diversi, viene applicato un default dipendente dal DBMS.
- › Lo standard per creare schemi consiste nell'utilizzo dell'istruzione SQL CREATE SCHEMA (con il simmetrico DROP SCHEMA).
- › Tuttavia, i DBMS di solito non supportano questa sintassi standard, ma preferiscono quella proprietaria!



Creazione di Database con MySQL

- › I comandi SQL usati per manipolare gli schemi in MySQL sono i seguenti:
 - **CREATE DATABASE [IF NOT EXISTS] *n***
crea il database *n*.
 - › La clausola IF NOT EXISTS permette di saltare senza errori questo comando se il database è già stato creato.
 - **DROP DATABASE *n***
elimina il database *n*
 - **USE *n***
seleziona il database *n*.
 - › Tutti i riferimenti a nomi (tabelle, procedure, ecc.) che non specificano un prefisso di database saranno considerati relativi al database selezionato.
- › Nella creazione del database è possibile specificare anche un set di caratteri. Tuttavia non ci occuperemo di questa caratteristica avanzata.
- › MySQL permette di usare la parola chiave SCHEMA al posto di DATABASE, per essere maggiormente allineato con lo standard.



Esempi

```
//iniziamo a creare un database per i nostri esempi...
```

```
CREATE DATABASE testcorso;
```

```
USE testcorso;
```



Creazione di Tabelle

- › La creazione di una tabella avviene tramite il comando CREATE TABLE, che ha la seguente sintassi:
 - **CREATE TABLE *nome_tabella*(*colonna*,...,*colonna*, *vincolo_tabella*,..., *vincolo_tabella*)**
 - › *nome_tabella* è il nome della tabella che viene creata
 - › *colonna* è una specifica di colonna
 - › *vincolo_tabella* è la specifica di un vincolo per le righe della tabella
- › Dopo la creazione, la tabella può essere rinominata o cancellata:
 - **DROP TABLE *nome_tabella***
cancella la tabella *nome_tabella*.
 - **RENAME *nome_tabella* TO *nuovo_nome_tabella***
rinomina la tabella *nome_tabella* in *nuovo_nome_tabella*.



Creazione di tabelle

Specificazione delle colonne

- › Il formato di una specifica di colonna è il seguente: ***nome_colonna dominio [DEFAULT valore] [vincolo_colonna ... vincolo_colonna]***
 - *nome_colonna* è il nome della colonna che viene creata
 - *dominio* è un tipo base di SQL o il nome di un dominio creato dall'utente
 - **DEFAULT** *valore* permette di dichiarare il valore di default per la colonna
 - *vincolo_colonna* è la specifica di un vincolo per la colonna. Vedremo tra poco quali vincoli possono essere espressi in questo caso.



Esempi

```
CREATE TABLE impiegati (  
    ID integer,  
    nome char(20),  
    cognome char(20),  
    mansione char(10),  
    data_assunzione date,  
    stipendio decimal(7,2),  
    premio decimal(7,2) DEFAULT 0.00,  
    IDreparto integer,  
    IDsuperiore integer  
);
```

```
CREATE TABLE impiegati (  
    ID integer,  
    nome char(20) ,  
    cognome char(20),  
    mansione enum('impiegato','dirigente') DEFAULT 'impiegato',  
    data_assunzione date,  
    stipendio decimal(7,2),  
    premio decimal(7,2) DEFAULT 0.00,  
    IDreparto integer,  
    IDsuperiore integer  
);
```

```
CREATE TABLE reparti (  
    ID int,  
    nome char(20)  
);
```



Gestione dei Permessi

- › Dopo aver creato degli utenti, si possono assegnare loro dei privilegi:
GRANT *privilegi* ON *elemento* TO *utente*
 - *privilegi* è una lista (separata da virgole) di privilegi. I privilegi disponibili sono identificati dalle parole chiave del linguaggio che permettono di utilizzare. Tra questi, citiamo: SELECT (lettura), INSERT (scrittura nuovi record), UPDATE (aggiornamento record), DELETE (cancellazione record), CREATE (creazione tabelle e database), CREATE VIEW (creazione di viste), ALTER (modifica di tabelle, viste, ecc.), DROP (cancellazione di tabelle, viste, ecc.)
 - › Si può utilizzare lo speciale privilegio ALL per assegnare tutti i privilegi disponibili
 - › E' inoltre possibile specificare se l'utente potrà propagare i suoi permessi ad altri utenti usando la sintassi **WITH GRANT OPTION**
 - L'*elemento* su cui si assegnano i privilegi può essere espresso in uno dei seguenti modi:
 - › *.* (tutte le tabelle di tutti gli schemi, privilegio globale)
 - › d.* (tutte le tabelle del database d)
 - › d.t (la tabella t del database d)
 - L'*utente* a cui si assegnano i privilegi deve essere specificato come richiesto dallo specifico DBMS.
- › Per rimuovere dei privilegi, si può usare il comando
REVOKE *privilegi* ON *elemento* FROM *utente*



Esempi

```
CREATE USER 'testuser1'@'localhost' IDENTIFIED BY 'testpass1';
```

```
GRANT CREATE, ALTER, SELECT, INSERT, UPDATE, DELETE ON testcorso.impiegati to  
'testuser1'@'localhost';
```

```
GRANT ALL ON testcorso.impiegati TO 'testuser1'@'localhost';
```

```
REVOKE ALL ON testcorso.* FROM 'testuser1'@'localhost';
```

```
GRANT CREATE, ALTER, SELECT, INSERT, UPDATE, DELETE ON testcorso.* to  
'testuser1'@'localhost';
```

```
GRANT ALL ON testcorso.impiegati TO 'testuser1'@'localhost';
```



Modifica di tabelle

- › Per modificare le colonne e i vincoli di una tabella sono disponibili i seguenti comandi:
 - **ALTER TABLE *nome_tabella* ADD COLUMN *colonna***
aggiunge una nuova *colonna* alla tabella *nome_tabella*.
 - **ALTER TABLE *nome_tabella* DROP COLUMN *nome_colonna***
rimuove la colonna *nome_colonna* dalla tabella *nome_tabella*.
 - **ALTER TABLE *nome_tabella* ALTER COLUMN *nome_colonna* SET DEFAULT *valore***
modifica il default della colonna *nome_colonna* nella tabella *nome_tabella*, impostandolo al *valore*.
 - **ALTER TABLE *nome_tabella* ALTER COLUMN *nome_colonna* DROP DEFAULT**
elimina il valore di default della colonna *nome_colonna* nella tabella *nome_tabella*.



Esempi

```
ALTER TABLE impiegati ADD COLUMN ufficio integer;
```

```
ALTER TABLE impiegati DROP COLUMN stipendio;
```

```
ALTER TABLE impiegati ALTER COLUMN ufficio SET DEFAULT 0;
```

```
ALTER TABLE impiegati ALTER COLUMN ufficio DROP DEFAULT;
```



Vincoli di integrità

- › Un vincolo è una regola che specifica delle condizioni sui valori delle colonne in una tabella.
- › Un vincolo può essere associato a una tabella o a un singolo attributo.
- › In SQL è possibile specificare diversi tipi di vincoli:
 - **Chiavi** (UNIQUE e PRIMARY KEY)
 - **Obbligatorietà di attributi** (NOT NULL)
 - **Chiavi esterne** (FOREIGN KEY)
 - **Vincoli generali** (CHECK)
- › È possibile specificare se il controllo del vincolo debba essere compiuto non appena si esegue un'operazione che ne può causare la violazione (**NOT DEFERRABLE**) o se *possa* essere rimandato alla fine della transazione (**DEFERRABLE**).
- › Per i vincoli differibili, è possibile modificare caso per caso la differibilità: la specifica **INITIALLY DEFERRED** (default) o **INITIALLY IMMEDIATE** indica il trattamento di default.
- › I vincoli non possono contenere condizioni la cui valutazione può dare risultati differenti a seconda momento in cui sono esaminati (es. riferimenti al tempo di sistema).



Attribuire un nome ai vincoli

- › Ogni vincolo ha associato nel DBMS un descrittore costituito da:
 - nome (se non specificato è assegnato automaticamente dal sistema)
 - specifica di differibilità
 - check time di default (per i vincoli differibili).
- › È possibile assegnare un nome ai vincoli facendo precedere la specifica del vincolo stesso dalla parola chiave **CONSTRAINT** e dal nome.
 - Specificare un nome per i vincoli è utile per potervisi riferire in seguito (ad esempio per modificarli o cancellarli).



Chiavi

- › La specifica delle chiavi si effettua in SQL mediante le parole chiave **UNIQUE** o **PRIMARY KEY**:
 - **UNIQUE** garantisce che non esistano due record che condividono gli stessi valori non nulli per gli attributi specificati (colonne **UNIQUE** possono contenere valori nulli, e in tal caso il vincolo non viene controllato).
 - **PRIMARY KEY** impone che per ogni record i valori degli attributi specificati siano non nulli e diversi da quelli di ogni altro record.
- › In una tabella è possibile specificare più chiavi **UNIQUE** ma una sola **PRIMARY KEY** (che andrebbe *sempre* specificata).
- › Se la chiave è formata da un solo attributo è sufficiente far seguire la specifica dell'attributo dalla parola chiave **UNIQUE** o **PRIMARY KEY** (*vincolo su attributo*).
- › Alternativamente si può far seguire la definizione della tabella dal *vincolo su tabella* **PRIMARY KEY(lista_colonne)** o **UNIQUE(lista_colonne)**, dove *lista_colonne* è una lista, separata da virgole, dei nomi delle colonne coinvolte nella chiave.



Chiavi primarie

- › Come sappiamo, **tutte le tabelle relazionali dovrebbero avere una chiave primaria**, che ne rende possibile la manipolazione dei singoli record.
 - Senza una chiave primaria, potrebbe essere impossibile identificare un preciso record della tabella, ad esempio per cancellarlo!
- › In molti casi alcune colonne di una tabella possono formare una chiave primaria, tuttavia è sempre opportuno *per motivi di ottimizzazione* **prediligere chiavi primarie su singole colonne numeriche**, usando invece un vincolo UNIQUE sulla chiave «naturale» della stessa tabella.
- › Come realizzare una chiave primaria «sintetica» di questo tipo? **Di solito si usa una colonna extra di tipo INTEGER**, che noi chiameremo sempre «ID» per chiarezza, e le si assegna **un valore unico per ciascun record** inserito nella tabella.
- › Molti database, tra cui MySQL, hanno un costrutto speciale che permette di **generare automaticamente un valore intero incrementale** per la colonna che costituisce la chiave primaria: nel caso di MySQL, basta far seguire alla specifica della colonna la parola chiave **auto_increment**.



Obbligatorietà di attributi (valori null)

- › Per indicare che un colonna non può assumere valore nullo (che è il default per tutti gli attributi di un record, se non specificato diversamente tramite la parola chiave DEFAULT usata nel dominio della colonna o sulla colonna stessa) è sufficiente includere il vincolo **NOT NULL** nella specifica della colonna.
- › Questo rende obbligatorio l'inserimento esplicito di un valore nella colonna per ogni record.



Esempi

```
CREATE TABLE impiegati (  
    ID integer NOT NULL PRIMARY KEY,  
    codice_fiscale char(16) UNIQUE,  
    nome char(20) NOT NULL,  
    cognome char(20) NOT NULL,  
    mansione enum('impiegato','dirigente') DEFAULT 'impiegato',  
    data_assunzione date,  
    stipendio decimal(7,2) NOT NULL,  
    premio decimal(7,2),  
    ufficio integer NOT NULL,  
    IDreparto integer NOT NULL,  
    IDsuperiore integer  
);
```

```
CREATE TABLE film (  
    titolo char(20) NOT NULL,  
    anno int(11) NOT NULL,  
    studio char(20),  
    colore tinyint(1),  
    PRIMARY KEY (titolo,anno)  
);
```



Chiavi esterne (foreign key)

- › In uno schema relazionale sono quasi sempre presenti riferimenti tra tabelle: i record di una tabella vengono messi in corrispondenza con i record di un'altra tabella incorporando nei primi la chiave primaria dei secondi.
- › SQL permette di codificare questi riferimenti nei metadati del database, in modo che il DBMS possa verificarne la sussistenza (integrità referenziale)
- › Una chiave esterna specifica che i valori di un particolare insieme di attributi (la chiave esterna) di ogni record devono necessariamente corrispondere a quelli presenti in un corrispondente insieme di attributi che sono una chiave per i record di un'altra tabella.
- › Una tabella può avere zero o più chiavi esterne. La specifica di chiavi esterne avviene mediante la clausola FOREIGN KEY inserita come vincolo su tabella:
FOREIGN KEY (lista_colonne1) REFERENCES nome_tabella(lista_colonne2)
[MATCH FULL | PARTIAL | SIMPLE]
[ON DELETE NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT]
[ON UPDATE NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT]
 - lista_colonne1 è un sottoinsieme delle colonne della tabella corrente, detta *referente*
 - lista_colonne2 è un insieme di colonne che costituisce una chiave per la tabella nome_tabella, detta *riferita*
 - Non è necessario che i nomi delle colonne messe in corrispondenza siano gli stessi, ma i domini degli attributi corrispondenti devono essere *compatibili*.
 - Nel caso di chiave esterna costituita da un solo attributo si può far semplicemente seguire la specifica dell'attributo da REFERENCES nome_tabella(nome_colonna) (vincolo su attributo). *Questa caratteristica non è supportata da MySQL.*
- › Il DBMS impedirà ogni inserimento o aggiornamento di record nella tabella referente che violi l'integrità referenziale, cioè produca record che non puntano correttamente a corrispondenti record nella tabella riferita.



Chiavi esterne

Tipi di Match

- › Il tipo di match è significativo nel caso di chiavi esterne costituite da più di un attributo e in presenza di valori nulli.
- › **MATCH SIMPLE:** il vincolo di integrità referenziale è soddisfatto se per ogni record della tabella referente
 - almeno una delle colonne della chiave esterna è null, oppure
 - nessuna di tali colonne è null ed esiste un record nella tabella riferita in cui i valori delle corrispondenti colonne coincidono con i valori delle colonne della chiave esterna.
- › **MATCH FULL**
 - tutte le colonne della chiave esterna sono null, oppure
 - nessuna di tali colonne è null ed esiste un record nella tabella riferita in cui i valori delle corrispondenti colonne coincidono con i valori delle colonne della chiave esterna.
- › **MATCH PARTIAL**
 - i valori delle colonne non nulle della chiave esterna corrispondono ai valori delle corrispondenti colonne in un record della tabella riferita.
- › Il default è MATCH SIMPLE.



Chiavi esterne

Azioni

- › Le clausole opzionali ON DELETE e ON UPDATE indicano al DBMS cosa fare nella tabella reference nel caso in cui un record cui si fa riferimento nella tabella riferito venga cancellato o modificato
- › Le azioni possibili per **ON DELETE** sono
 - **NO ACTION**
la cancellazione di un record dalla tabella riferita è eseguita solo se non esiste alcun record nella tabella referente che vi fa riferimento. In altre parole, l'operazione di cancellazione viene *respinta* se il record da cancellare è in relazione con qualche record della tabella referente.
 - **RESTRICT**
come per NO ACTION, con la differenza che questa condizione viene controllata subito, mentre NO ACTION viene considerata dopo che sono state esaminate tutte le altre specifiche relative all'integrità referenziale. *In MySQL NO ACTION RESTRICT sono sinonimi.*
 - **CASCADE**
la cancellazione di un record dalla tabella riferita implica la cancellazione di tutti i record della tabella referente che vi fanno riferimento
 - **SET NULL**
la cancellazione di un record dalla tabella riferita implica che, in tutti i record della tabella referente che vi fanno riferimento, la chiave esterna viene posta al valore null (se ammesso).
 - **SET DEFAULT**
la cancellazione di un record dalla tabella riferita implica che, in tutti i record della tabella referente che vi fanno riferimento, il valore della chiave esterna viene posto uguale al valore di default specificato per le colonne che la costituiscono.
- › Le azioni ON UPDATE, invece, vengono attivate quando la chiave di un record riferito tramite chiave esterna da un'altra tabella viene modificata.
 - Le azioni possibili sono le stesse, con l'eccezione di CASCADE, che ha l'effetto di assegnare alla chiave esterna il nuovo valore della chiave del record riferito.
- › Il default è NO ACTION sia per la cancellazione sia per la modifica.
- › L'ordine in cui vengono considerate le varie opzioni (nel caso di più riferimenti) è RESTRICT, CASCADE, SET null, SET DEFAULT, NO ACTION.



Esempi

```
CREATE TABLE impiegati (  
  ID integer NOT NULL PRIMARY KEY,  
  codice_fiscale char(16) UNIQUE,  
  nome char(20) NOT NULL,  
  cognome char(20) NOT NULL,  
  mansione enum('impiegato','dirigente') DEFAULT 'impiegato',  
  data_assunzione date,  
  stipendio decimal(7,2) NOT NULL,  
  premio decimal(7,2),  
  ufficio integer NOT NULL,  
  IDreparto int NOT NULL,  
  IDsuperiore integer,  
  FOREIGN KEY (IDsuperiore) REFERENCES impiegati(ID) ON UPDATE CASCADE ON DELETE SET  
  NULL  
);
```

//notare la foreign key ciclica: IDsuperiore punta a un altro elemento della stessa tabella, che rappresenta il "superiore" dell'impiegato. Questo valore può essere null (per chi non ha un superiore) e lo diventa in ogni caso quando il superiore viene cancellato (ON DELETE).

```
CREATE TABLE reparti (  
  ID int NOT NULL PRIMARY KEY,  
  nome char(20) NOT NULL,  
  indirizzo char(200) NOT NULL,  
  citta char(100) NOT NULL  
);
```

//aggiungiamo la foreign key che segue alla tabella impiegati in un secondo momento, oppure definiamo reparti prima di impiegati

```
ALTER TABLE impiegati ADD CONSTRAINT appartenenza FOREIGN KEY (IDreparto)  
REFERENCES reparti(ID) ON UPDATE CASCADE;
```



Esempi

```
CREATE TABLE docente (  
    ID int(11) NOT NULL,  
    nome varchar(20),  
    residenza varchar(15),  
    PRIMARY KEY (ID)  
);
```

```
CREATE TABLE studente (  
    ID int(11) NOT NULL,  
    nome varchar(20),  
    residenza varchar(15),  
    nascita date DEFAULT NULL,  
    PRIMARY KEY (ID)  
);
```

```
CREATE TABLE relatore (  
    IDdocente int(11) NOT NULL,  
    IDstudente int(11) NOT NULL,  
    PRIMARY KEY (IDdocente, IDstudente),  
    FOREIGN KEY (IDstudente) REFERENCES studente (ID) ON DELETE CASCADE ON  
UPDATE CASCADE  
    FOREIGN KEY (IDdocente) REFERENCES docente (ID) ON DELETE RESTRICT ON  
UPDATE CASCADE  
);
```



Vincoli CHECK

- › Quando si definisce una tabella è possibile aggiungere alla specifica di ciascuna colonna la parola chiave **CHECK** seguita da una condizione, cioè un predicato o una combinazione booleana di predicati, comprendenti anche sottointerrogazioni che fanno riferimento ad altre tabelle (componente WHERE di una query SQL).
- › Il vincolo viene controllato solo quando viene inserito o modificato un record nella tabella.
- › I vincoli check possono essere anche scritti come vincoli su tabella (dopo la dichiarazione di tutte le colonne).
- › **Sono supportati da MySQL solo a partire dalla versione 8.0.16.**
 - In MySQL, i vincoli check devono contenere espressioni semplici, ad esempio non possono fare riferimento a funzioni utente, colonne auto_increment e funzioni interne non deterministiche. Non si possono inserire query nelle espressioni CHECK.



Esempi

```
CREATE TABLE impiegati (  
premio decimal(7,2),  
stipendio decimal(7,2),  
CHECK (stipendio > premio)  
);
```

```
CREATE TABLE impiegati (  
stipendio decimal(7,2),  
CHECK (stipendio < (select max(stipendio) from impiegati where mansione  
='dirigente')) //NON supportato da MySQL  
);
```

```
CREATE TABLE impiegati (  
mansione char(10),  
CHECK (mansione in ('dirigente','ingegnere','tecnico','segretaria'))  
);
```




Valutazione dei vincoli

- › Nel valutare i vincoli, il DBMS si attiene alle seguenti regole:
 - Un vincolo su un record è violato se la condizione valutata sul record restituisce valore FALSE.
 - Un vincolo la cui valutazione su un dato record restituisce valore TRUE o UNKNOWN (a causa di valori nulli) non è violato.
- › *Durante l'inserimento o la modifica di un insieme di record, la violazione di un vincolo per uno dei record causa la non esecuzione dell'intero insieme.*



Modificare i vincoli di una tabella

- › Per modificare un vincolo è necessario conoscerne il nome.
 - **ALTER TABLE *nome_tabella* DROP CONSTRAINT *c* RESTRICT | CASCADE**
cancella il vincolo chiamato *c* dalla tabella *nome_tabella*.
 - **ALTER TABLE *nome_tabella* ADD [CONSTRAINT *c*] *vincolo_colonna***
crea un vincolo chiamato *c* (o anonimo) nella tabella *nome_tabella*.
vincolo_colonna può essere uno qualsiasi dei vincoli visti in precedenza.
 - **ALTER TABLE *nome_tabella* SET CONSTRAINTS (*lista_vincoli* | ALL) IMMEDIATE | DEFERRED**
modifica il check time di default (solo per i vincoli DEFERRABLE) di tutti i vincoli o di quelli specificati.



Modificare i vincoli di una tabella MySQL

- › MySQL non dispone del comando DROP CONSTRAINT, ma predilige una sintassi specifica per il vincolo da rimuovere
 - Per rimuovere la chiave primaria:
ALTER TABLE *nome_tabella* DROP PRIMARY KEY
 - Per rimuovere un vincolo UNIQUE chiamato *c*, ci si basa sul fatto che questo crea implicitamente un indice indici, e si usa quindi la sintassi
ALTER TABLE *nome_tabella* DROP INDEX *c*
 - Per rimuovere un vincolo check:
ALTER TABLE *nome_tabella* DROP CHECK *c*
 - *Per modificare altri vincoli su colonna (ad esempio un NOT NULL) è necessario ridefinire la colonna con un ALTER COLUMN*
 - Per rimuovere una chiave esterna *c*, si usa la sintassi
ALTER TABLE *nome_tabella* DROP FOREIGN KEY *c*



Esempi

```
ALTER TABLE impiegati ADD UNIQUE (nome, cognome);
```

```
ALTER TABLE impiegati ADD CONSTRAINT stipok CHECK (stipendio < premio);
```

SQL: DML

*Inserimento, aggiornamento ed
eliminazione dei dati*





Modifica dei Dati (DML)

- › In SQL esistono tipi di query che permettono di modificare i dati nelle tabelle:
 - Query di *inserimento*
 - Query di *aggiornamento*
 - Query di *cancellazione*
- › Questi tre tipi di query, che saranno descritte più in dettaglio nelle sezioni che seguono, integrano nella loro sintassi quella delle più generali query di selezione.
- › Le query di modifica vengono comunque trattate prima di quelle di selezione, poichè il loro uso è necessario per popolare la base di dati prima di effettuare interrogazioni.



Query di inserimento

- › Le query di inserimento permettono di inserire nuovi record in una tabella.
 - L'origine dei record può essere una serie di valori costanti forniti dall'utente o un insieme di record restituito da una query di selezione.
 - E' possibile specificare quali attributi verranno inseriti nel record e in che ordine. Se non vengono specificati tutti gli attributi, gli altri avranno il loro valore di default (sempre che ciò non violi qualche vincolo).
- › **INSERT INTO *nome_tabella* [(*lista_colonne*)] VALUES (*lista_valori*)**
inserisce un nuovo record nella tabella *nome_tabella*.
 - Nel nuovo record, il valore di ciascuna colonna *c*, contenuta nella *lista_colonne*, viene impostato al corrispondente valore *v* inserito nella *lista_valori* (i tipi devono ovviamente essere compatibili).
 - Se non si specifica una lista di nomi di colonne, la lista dei valori dovrà contenere un valore per ciascuna colonna della tabella, esattamente nell'ordine con cui le colonne sono state create (tramite il comando CREATE).
- › **INSERT INTO *nome_tabella* [(*lista_colonne*)] query**
inserisce nella tabella *nome_tabella* tanti record quante sono le righe restituite dalla query di selezione specificata.
 - La query deve restituire una tabella formata da tante colonne quante sono quelle specificate nella *lista_colonne*, con tipi corrispondenti compatibili.
 - Se si omette la *lista_colonne*, la query dovrà restituire una tabella compatibile (per colonne, ordine delle stesse e tipi) con *nome_tabella*.



Esempi

```
INSERT INTO reparti(ID,nome) VALUES (0,'Amministrazione')  
//inserisce un record nella tabella reparti.
```

```
INSERT INTO  
impiegati(ID,nome,cognome,stipendio,premio,IDreparto,mansione,ufficio) VALUES  
(235,'Mario','Rossi',1,2,0,'dirigente',1)  
//inserisce un record nella tabella impiegati.
```

```
INSERT INTO  
impiegati(ID,nome,cognome,stipendio,premio,IDreparto,mansione,ufficio) VALUES  
(236,'Carlo','Bianchi',1,2,1,'impiegato',1)  
//inserisce un record nella tabella impiegati. Se la foreign key definita  
negli esempi precedenti è presente, questo inserimento fallisce
```

```
INSERT INTO mansioni SELECT DISTINCT mansione FROM impiegati  
//inserisce tutti i record derivanti dalla query SELECT DISTINCT mansione  
FROM impiegati (vedi dopo) nella tabella mansioni.
```




Query di aggiornamento

- › Le query di aggiornamento permettono di variare il contenuto dei record di una tabella del database che corrispondono a un certo criterio.
 - I record da modificare vengono selezionati tramite una clausola WHERE
 - A tutti i record selezionati vengono applicati degli aggiornamenti che possono essere costanti o far riferimento agli attributi dello stesso record.
- › **UPDATE *nome_tabella* SET *nome_colonna* = *espressione*, ... WHERE *condizione***
aggiorna la tabella *nome_tabella* impostando, per ogni record selezionato dalla *condizione*, ciascuna *nome_colonna* specificata al valore calcolato dalla corrispondente *espressione*.
 - *nome_colonna* è il nome di una colonna della tabella specificata
 - *espressione* è un'espressione costante, o che fa riferimento agli altri attributi della tabella specificata, in cui si possono utilizzare tutti gli operatori (aritmetici, stringa, ...) messi a disposizione dal DBMS
 - *condizione* è una condizione WHERE standard (vedi query di selezione).



Query di cancellazione

- › Con una query di cancellazione si possono cancellare da una tabella i record che corrispondono a un certo criterio (espresso con una clausola WHERE), o tutti i record.
- › **DELETE FROM *nome_tabella* [WHERE *condizione*]**
cancella dalla tabella *nome_tabella* i record che corrispondono alla *condizione* data.
 - Omettendo la *condizione*, l'intero contenuto della tabella verrà eliminato.



Esempi

```
UPDATE impiegati SET stipendio = stipendio+stipendio*0.1, premio = 1000 WHERE  
mansione = 'dirigente'
```

//incrementa lo stipendio del 10% e imposta il premio a 1000 in tutti i
record della tabella impiegati la cui mansione è 'dirigente'.

```
DELETE FROM impiegati WHERE data_assunzione > DATE('1/1/2001')
```

//elimina dalla tabella impiegati tutti i record in cui DATA_A è successiva
al 1/1/2001.

```
DELETE FROM impiegati
```

//svuota la tabella impiegati.

SQL: QL

Interrogazioni sui dati





Interrogazioni (QL)

- › Le interrogazioni, o query di selezione, sono la funzionalità principale di SQL.
- › Tramite le interrogazioni è possibile estrarre dati dal database secondo criteri anche molto complessi, generando in output
 - **tabelle**: si tratta della forma di risultato più generale e conosciuta. Le tabelle restituite da un'interrogazione sono temporanee, cioè non sono inserite nella base di dati, ma possono essere usate per creare *viste* sui dati.
 - **liste**: questo tipo di interrogazione restituisce una tabella formata da una singola colonna.
 - › In alcune espressioni SQL questo tipo di risultato può essere utilizzato come una lista di valori (ad esempio, come secondo operando dell'istruzione IN).
 - **valori singoli**: questo tipo di interrogazione, detta *singleton query*, restituisce una tabella formata da un solo record con una sola colonna.
 - › Il valore così restituito può anche essere riusato in molte espressioni SQL come fosse il risultato di una speciale chiamata di funzione.



Interrogazioni di base

- › La sintassi di base di una interrogazione SQL è la seguente:

SELECT *lista_attributi* FROM *lista_tabelle* [WHERE *condizione*]

- la clausola SELECT dichiara le colonne da estrarre, il loro ordine e il loro nome.
 - la clausola FROM dichiara le tabelle dalle quali verranno estratti i dati.
 - la clausola opzionale WHERE definisce una condizione di filtro per i record da estrarre.
- › Dal punto di vista del DBMS, la query viene eseguita secondo la seguente procedura:
 1. Si genera il **prodotto cartesiano** di tutte le tabelle specificate dalla clausola FROM, nell'ordine dato.
 2. Si applica la **condizione** di ricerca specificata nella clausola WHERE ai record del prodotto cartesiano generato al passo precedente.
 3. Si restituiscono in **output** i valori delle colonne specificate dalla clausola SELECT.



Interrogazioni di base

Clausola FROM

- › La clausola FROM dichiara le tabelle dalle quali verranno estratti i dati.
- › La *lista_tabelle* ha la forma
***nome_tabella* [[AS] [*alias*]], *nome_tabella* [[AS] [*alias*]], ...**
 - Gli alias possono essere usati per rinominare una tabella localmente alla query. In tutte le altre clausole si dovrà utilizzare l'*alias* invece del *nome_tabella* originale.
 - › Questo è indispensabile nel caso si importino più istanze della stessa tabella all'interno della query.
- › Sia la condizione di filtro (WHERE) che le colonne indicate nella clausola SELECT possono far riferimento alle sole tabelle importate tramite la clausola FROM, utilizzando eventualmente l'alias dichiarato come loro nome.
- › In assenza di direttive specifiche, la query opererà sul prodotto cartesiano delle tabelle indicate, nell'ordine dato.



Interrogazioni di base

Clausola WHERE

- › La condizione espressa dopo la parola chiave WHERE è un'espressione che deve avere valore booleano (true/false/unknown).
 - Per la composizione di questa espressione si possono usare tutti gli operatori e le funzioni visti in precedenza.
- › I record selezionati dalla query sono tutti e soli quelli per cui la condizione è true.



Interrogazioni di base

Clausola SELECT

- › la clausola SELECT dichiara le colonne da estrarre da ciascun record selezionato, il loro ordine e il loro nome.
- › Può essere anche usata per estrarre **colonne calcolate** sulla base di dei valori di altre colonne, o persino valori del tutto scorrelati dai record estratti.
- › La *lista_attributi* ha la forma
***nome_colonna* [[AS] [*alias*]],*nome_colonna* [[AS] [*alias*]], ...**
 - Ogni colonna è indicata semplicemente col suo nome nel caso questo non sia ambiguo (cioè se nell'interrogazione non sono presenti più tabelle che hanno colonne con lo stesso nome). In caso di ambiguità, è possibile usare la notazione *nome_tabella.nome_colonna* per indicare la tabella specifica contenente la colonna desiderata (qui *nome_tabella* può anche essere un alias definito nella clausola FROM).
 - Il simbolo speciale * usato in questa clausola indica ad SQL di estrarre tutte le colonne. E' possibile anche scrivere *nome_tabella.** per estrarre tutte le colonne di una particolare tabella.
 - Tramite gli alias è possibile ridefinire il nome delle colonne estratte, o fornire un nome alle colonne calcolate, che non ne hanno uno predefinito.



Esempi: Dati

ID	Nome	Cognome	Stipendio	Data_assunzione	Ufficio	IDreparto	IDsuperiore	Mansione
1	Mario	Rossi	45	1/1/2000	10	1		dirigente
2	Carlo	Bianchi	36	1/7/2000	20	2		impiegato
3	Giuseppe	Verdi	40	6/5/2005	20	1	1	impiegato
4	Franco	Neri	45	4/2/2008	16	3		impiegato
5	Carlo	Rossi	80	12/11/2008	14	4		dirigente
6	Lorenzo	Lanzi	73	24/8/2004	7	4		dirigente
7	Paola	Borroni	40	23/12/2003	75	1	1	impiegato
8	Marco	Franco	46	3/1/2000	29	2	2	impiegato

ID	Nome	Indirizzo	Città
1	Amministrazione	via Tito Livio, 27	Milano
2	Produzione	p.le Lavater, 3	Torino
3	Distribuzione	via Segre, 9	Roma
4	Direzione	via Tito Livio, 27	Milano
5	Ricerca	via Morone, 6	Milano



Esempi

```
//Reperire i cognomi degli impiegati del reparto 2
SELECT cognome FROM impiegati WHERE IDreparto = 2
//risultato: Bianchi e Franco

//Reperire tutti i dati degli impiegati del reparto 1
SELECT * FROM impiegati WHERE IDreparto = 1

//Reperire lo stipendio mensile degli impiegati di cognome Bianchi
SELECT stipendio/12 AS stipendio_mensile FROM impiegati WHERE cognome = 'BIANCHI'

//Reperire i cognomi degli impiegati del reparto 1 o 2
SELECT cognome FROM impiegati WHERE IDreparto=1 OR Idreparto=2

//Lista di tutti i manager
SELECT * FROM impiegati WHERE mansione = 'dirigente'

//Lista gli ID di tutti gli impiegati
SELECT ID FROM impiegati

//Lista di tutti i reparti con ID maggiore o uguale a 30
SELECT * FROM reparti WHERE ID>=30

//Listare gli impiegati che hanno un premio di produzione maggiore dello stipendio
SELECT * FROM impiegati WHERE premio > stipendio

//Selezionare la mansione degli impiegati che hanno uno stipendio maggiore di 50
SELECT mansione FROM impiegati WHERE stipendio > 50

//Selezionare il nome e il numero di reparto degli impiegati che hanno uno
stipendio maggiore di 50 e mansione 'impiegato'
SELECT nome, IDreparto FROM impiegati WHERE stipendio > 50 AND mansione
='impiegato'
```



Esempi

//Selezionare l'ID degli impiegati che lavorano nel dipartimento 1 e sono impiegati o dirigenti

```
SELECT ID FROM impiegati WHERE IDreparto=1 AND(mansione = 'impiegato' OR  
mansione = 'dirigente')
```

//Lista degli impiegati dei reparto 4 che guadagnano più di 30

```
SELECT cognome, stipendio FROM impiegati WHERE stipendio > 30 AND IDreparto=4
```

//Trovare il nome, stipendio e premio di produzione di tutti gli impiegati per cui la somma dello stipendio e dei premio di produzione è maggiore di 20

```
SELECT nome, stipendio, premio FROM impiegati WHERE mansione ='impiegato' AND  
stipendio+premio > 20
```

//Lista degli impiegati (con nome e stipendio) aventi stipendio compreso tra 20 e 40

```
SELECT nome, stipendio FROM impiegati WHERE stipendio BETWEEN 20 AND 40
```

//Tutti i dati dei reparti 1 e 3

```
SELECT * FROM reparto WHERE ID IN (1,3);
```

//Lista degli impiegati che non sono né dirigenti né impiegati

```
SELECT nome, stipendio, IDreparto, mansione FROM impiegati WHERE mansione NOT  
IN ('dirigente','impiegato')
```

//Determinare tutti gli impiegati il cui cognome comincia per 'R'

```
SELECT nome, cognome FROM impiegati WHERE cognome LIKE 'R%'
```



Esempi

```
//Trovare gli impiegati che non hanno un superiore  
SELECT cognome FROM impiegati WHERE IDsuperiore IS NULL
```

```
//Trovare il nome, lo stipendio, il premio di produzione, e la somma dello  
stipendio e del premio di produzione di tutti i dirigenti  
SELECT nome, stipendio, premio, stipendio+premio FROM impiegati WHERE mansione  
='dirigente'
```

```
//Generare una tabella che contenga lo stipendio corrente di ciascun impiegato,  
indicando la data in cui la tabella è stata generata.  
SELECT CURRENT_DATE, nome, stipendio FROM impiegati;  
//la prima colonna contiene una costante, che verrà ripetuta su ogni riga. E' un  
particolare tipo di colonna calcolata, utile in casi particolari, ad esempio quando  
si creano tabelle tramite unione.
```

```
//Restituire una lista di stringhe di testo nel formato "cognome nome - mansione"  
per ogni record nella tabella impiegati  
SELECT CONCAT(cognome , ' ' , nome , ' - ' , mansione) as stringa FROM impiegati
```

```
//Si vuole avere un colloquio con tutti i nuovi impiegati del reparto 1 dopo 90  
giorni dalla loro assunzione. In particolare, per tutti gli impiegati interessati,  
si vogliono determinare il nome, la data di assunzione e la data del colloquio  
SELECT nome, DATE_FORMAT(data_assunzione,'%d/%m/%Y') data_assunzione,  
DATE_FORMAT(DATE_ADD(data_assunzione, INTERVAL 90 DAY),'%d/%m/%Y') as  
data_colloquio FROM impiegati WHERE DATE_ADD(data_assunzione, INTERVAL 90 DAY) =  
CURRENT_DATE AND Idreparto=1;
```



Eliminazione dei duplicati

- › Può accadere che, in base alle colonne selezionate, una query ritorni più record con lo stesso contenuto. Ovviamente questo non è in generale utile.
- › In questo caso è possibile richiedere l'eliminazione dei duplicati tramite la parola chiave **DISTINCT** inserita nella clausola SELECT:
SELECT DISTINCT ... FROM ... [WHERE ...]
- › La parola chiave DISTINCT forza l'interrogazione a restituire record distinti, cioè privi di duplicati. Nel caso siano restituite righe con più colonne, l'intera riga viene utilizzata per determinare l'unicità.



Esempi

```
//Determinare tutte le città in cui è presente un reparto
```

```
SELECT citta FROM reparti
```

```
//la stessa città compare più volte se ospita più reparti
```

```
SELECT DISTINCT citta FROM reparti
```

```
//così ogni città compare una sola volta
```

```
//Trovare i cognomi più lunghi di tre caratteri nella tabella impiegati
```

```
SELECT DISTINCT cognome FROM impiegati WHERE LENGTH(cognome) > 3
```



Ordinamento del risultato di una query

- › Negli esempi visti finora l'ordine dei record risultato di una interrogazione è determinato dal sistema (dipende dalla strategia usata per eseguire l'interrogazione).
- › È possibile specificare un ordinamento diverso aggiungendo *alla fine dell'interrogazione* la clausola **ORDER BY**:
SELECT ... FROM ... [WHERE ...] [ORDER BY ...]
- › Dal punto di vista del DBMS, la query con questa nuova clausola viene eseguita secondo la seguente procedura:
 1. Si genera il **prodotto cartesiano** di tutte le tabelle specificate dalla clausola FROM, nell'ordine dato.
 2. Si applica la **condizione** di ricerca specificata nella clausola WHERE ai record della tabella generata al passo precedente.
 3. Si creano i **record di output** con i valori delle colonne/espressioni specificate dalla clausola SELECT calcolati sulla tabella filtrata ottenuta al passo precedente.
 4. Si **ordinano i record** ottenuti al passo precedente in base alla clausola ORDER BY.



Ordinamento del risultato di una query

Clausola ORDER BY

- › La clausola ORDER BY permette di specificare uno o più criteri di ordinamento:
- › **ORDER BY** *colonna* [ASC | DESC], *colonna* [ASC | DESC], ...
 - I dati saranno ordinati in base al contenuto delle colonne citate. Se si specifica più di una colonna, le successive saranno usate, nell'ordine, per ordinare le righe che risultino uguali secondo le colonne precedenti (ordinamento secondario).
 - Il metodo di ordinamento dipende dal tipo di dato della colonna, e di default l'ordine è discendente. E' possibile specificare le parole chiave ASC (ascendente) o DESC (discendente) per forzare un particolare ordinamento su ciascuna colonna indicata.
- › In alcuni DBMS è possibile **usare come criteri di ordinamento anche** nomi (alias) di **colonne calcolate** dalla clausola SELECT.



Esempi

```
//Elencare lo stipendio, la mansione e il nome di tutti gli impiegati del  
dipartimento 3, ordinando i risultati in ordine crescente di stipendio  
SELECT stipendio, mansione, nome  
FROM impiegati  
WHERE IDreparto = 3  
ORDER BY stipendio;
```

```
//Si vogliono elencare mansione, nome e stipendio di tutti gli impiegati  
ordinando i risultati in base alla mansione in ordine crescente, e poi in  
base allo stipendio in ordine decrescente  
SELECT mansione, stipendio, nome  
FROM impiegati  
ORDER BY mansione, stipendio DESC;
```

```
//Elencare lo stipendio complessivo (compreso il premio di produzione) di  
tutti gli impiegati, ordinando i risultati in ordine crescente di stipendio  
complessivo  
SELECT stipendio+premio AS stipendio_totale, nome  
FROM impiegati  
WHERE IDreparto = 0  
ORDER BY stipendio_totale ASC;
```



Join tra tabelle

- › Finora le query viste **hanno operato sul prodotto cartesiano delle tabelle coinvolte**, ma sappiamo dalla teoria relazionale che è possibile effettuare **join** diversi tra tabelle, che restituiscono **sottoinsiemi del prodotto cartesiano**.
- › In SQL, un join può essere effettuato implicitamente (tramite la clausola WHERE) o esplicitamente (tramite l'operatore JOIN).
- › Il join implicito può essere espresso in SQL tramite l'applicazione di uno o più **predicati di join inseriti nella clausola WHERE**, che filtrino un sottoinsieme dei record del un prodotto cartesiano generato dalla clausola FROM.
- › In pratica, la clausola WHERE viene usata in questo caso sia per filtrare il sottoinsieme del prodotto cartesiano di nostri interesse, che per (eventualmente) filtrare da questo sottoinsieme i record da estrarre effettivamente.



Esempi

```
//Determinare il nome del reparto in cui lavora l'impiegato Rossi
SELECT reparti.nome
FROM impiegati, reparti
WHERE impiegati.cognome = 'Rossi' AND impiegati.IDreparto = reparti.ID;
//Il predicato di join è impiegati.IDreparto=reparti.ID
//omettendolo avremmo in output il record relativo all'impiegato Rossi
combinato con TUTTI i record della tabella reparti, e quindi la SELECT
ritornerebbe i nomi di tutti i reparti.
```

```
//Lista di tutti gli impiegati che guadagnano più di Rossi (theta-join)
SELECT altri.cognome, altri.stipendio, altri.mansione
FROM impiegati rossi, impiegati altri
WHERE altri.stipendio > rossi.stipendio AND rossi.cognome='Rossi'
//questa interrogazione mostra anche come è possibile importare più di
un'istanza della stessa tabella in una query usando il costrutto degli alias.
```

```
//Lista degli impiegati con i rispettivi capi (self-join)
SELECT subordinato.cognome AS impiegato, capo.cognome AS superiore
FROM impiegati subordinato, impiegati capo
WHERE subordinato.IDsuperiore=capo.ID
//chi non ha capo (IDsuperiore IS NULL) non compare nella lista!
```



L'operatore JOIN

- › SQL:1999 prevede diversi tipi di operatori di join espliciti.
- › In pratica, questi operatori producono nuove tabelle temporanee, e quindi possono essere usati nella clausola FROM al posto dei semplici nomi di tabella.
- › Esistono (almeno) due tipi di operatore JOIN disponibili in ogni dialetto SQL
 - JOIN *interno*
 - JOIN *esterno*
- › L'operatore JOIN può essere usato per realizzare join tra tabelle altrimenti molto complessi o impossibili (come i join esterni) da realizzare usando il metodo implicito appena visto.
- › E' possibile mescolare, nella clausola FROM, nomi di tabella a espressioni di JOIN, ottenendo come risultato il prodotto cartesiano tra le tabelle reali e quelle temporanee generate dai join espliciti
- › E' possibile inoltre combinare i JOIN: i suoi due argomenti, infatti, possono essere tabelle reali o tabelle temporanee create, a loro volta, da altre operazioni di JOIN (in questo caso si possono usare le parentesi per rendere più chiare le espressioni).



Join interni (inner join)

- › I join interni sono i più comuni, e restituiscono il sottoinsieme del prodotto cartesiano tra le due tabelle che rispetta certe condizioni.
- › Sono quelli più facilmente riproducibili con il metodo implicito (condizioni WHERE), ma realizzarli con l'operatore esplicito presenta vantaggi in termini di leggibilità e, a seconda del DBMS, anche di velocità.
- › ***tabella1* CROSS JOIN *tabella2***
genera il *cross join* tra *tabella1* e *tabella2*, cioè il loro prodotto cartesiano
 - Solitamente omissivo, visto che il prodotto cartesiano è il default se non si specifica un operatore di JOIN tra le tabelle della clausola FROM
- › ***tabella1* NATURAL JOIN *tabella2***
genera il join naturale tra *tabella1* e *tabella2*. Le due tabelle devono avere una o più colonne con lo stesso nome.
 - Il risultato contiene solo le coppie di record (r1,r2) presi dalle due tabelle per i quali tutte le colonne omonime hanno lo stesso valore
- › ***tabella1* [INNER] JOIN *tabella2* USING (*lista_colonne*)**
genera il join naturale tra *tabella1* e *tabella2* usando le colonne specificate nella *lista_colonne* (che devono essere presenti in entrambe le tabelle).
 - Il risultato contiene solo le coppie di record (r1,r2) presi dalle due tabelle per i quali le colonne omonime in *lista_colonne* hanno lo stesso valore
- › ***tabella1* [INNER] JOIN *tabella2* ON condizione**
genera il *theta join* tra *tabella1* e *tabella2* secondo la condizione specificata.
 - Il risultato contiene solo le coppie di record (r1,r2) presi dalle due tabelle per i quali la condizione è vera



Join esterni (outer join)

- › Nel normale join tra due tabelle R e S non si ha traccia dei record di R che non corrispondono ad alcun record di S. Questo non sempre è quello che si desidera.
- › L'operatore di OUTER JOIN aggiunge al risultato i record di R e S che non hanno partecipato al join, **completandoli con colonne null**.
 - ***tabella1* LEFT [OUTER] JOIN *tabella2* ON *condizione***
fornisce come risultato il theta join interno tra *tabella1* e *tabella2* esteso con le righe della tabella che compare a sinistra del join (*tabella1*) per le quali non esiste una corrispondente riga nella tabella di destra.
 - ***tabella1* RIGHT [OUTER] JOIN *tabella2* ON *condizione***
fornisce come risultato il theta join interno tra *tabella1* e *tabella2* esteso con le righe della tabella che compare a destra del join (*tabella2*) per le quali non esiste una corrispondente riga nella tabella di sinistra.
 - ***tabella1* FULL [OUTER] JOIN *tabella2* ON *condizione***
fornisce come risultato il theta join interno tra *tabella1* e *tabella2* esteso con le righe delle due tabelle che non hanno un corrispondente secondo la condizione di join.
 - › Non è supportato da MySQL
- › Tutti i JOIN appena visti hanno anche la variante **USING (*lista_colonne*)** al posto di **ON *condizione***, in cui la condizione di join è l'eguaglianza dei valori nelle colonne di *lista_colonne*, che devono essere presenti in entrambe le tabelle.
- › La variante OUTER può essere utilizzata anche per il join naturale, escludendo la clausola ON (**NATURAL LEFT [OUTER] JOIN**, **NATURAL RIGHT [OUTER] JOIN**)



Esempi

```
//Selezionare gli impiegati e la città in cui lavorano.  
SELECT impiegati.nome, impiegati.cognome, reparti.citta  
FROM impiegati INNER JOIN reparti ON (impiegati.IDreparto=reparti.ID)
```

```
//Lista degli impiegati con i rispettivi capi  
SELECT subordinato.cognome AS impiegato, capo.cognome AS superiore  
FROM impiegati subordinato JOIN impiegati capo ON  
(subordinato.IDsuperiore=capo.ID)  
//chi non ha capo (IDsuperiore IS NULL) non compare nella lista!
```

```
//Lista di TUTTI impiegati con i relativi capi, se presenti  
SELECT subordinato.cognome AS impiegato, capo.cognome AS superiore  
FROM impiegati subordinato LEFT JOIN impiegati capo ON  
(subordinato.IDsuperiore=capo.ID)  
//anche chi non ha capo (IDsuperiore IS NULL) compare nella lista
```

```
//tutte le persone che hanno una casa  
SELECT * FROM persona JOIN casa ON (persona.IDcasa = casa.ID)  
//tutte le persone e le case eventualmente ad esse associate  
SELECT * FROM persona LEFT JOIN casa ON (persona.IDcasa = casa.ID)  
//tutte le case e le persone che eventualmente le posseggono  
SELECT * FROM persona RIGHT JOIN casa ON (persona.IDcasa = casa.ID)  
//tutte le persone e le case, associate quando possibile  
SELECT * FROM persona FULL JOIN casa ON (persona.IDcasa = casa.ID)
```




Aggregazione di Record

- › In algebra relazionale, le condizioni sono predicati valutati su singole tuple indipendentemente dalle altre. Spesso è invece necessario estrarre o valutare proprietà che dipendono da insiemi di record.
 - Ad esempio, il «numero di impiegati del reparto Produzione» non è una proprietà di un singolo record.
- › Gli **operatori aggregati** di SQL permettono di calcolare valori aggregando più record secondo particolari criteri:
 - **COUNT (*)**
conta il **numero di record** selezionati.
 - **COUNT ([DISTINCT | ALL] *espressione*)**
conta il **numero di valori non nulli** calcolati dall'espressione su ciascuno dei record selezionati.
 - **SUM ([DISTINCT | ALL] *espressione*)**
somma i valori dell'espressione calcolati su ciascuno dei record selezionati.
 - **MAX ([DISTINCT | ALL] *espressione*)**
calcola il **massimo** tra i valori dell'espressione calcolati su ciascuno dei record selezionati.
 - **MIN ([DISTINCT | ALL] *espressione*)**
calcola il **minimo** tra i valori dell'espressione calcolati su ciascuno dei record selezionati.
 - **AVG ([DISTINCT | ALL] *espressione*)**
calcola la **media** tra i valori dell'espressione calcolati su ciascuno dei record selezionati.
- › In tutti i casi, l'*espressione* può essere una qualsiasi tra quelle ammissibili all'interno di una clausola SELECT: dal nome di una colonna a una formula complessa.
- › I modificatori ALL e DISTINCT possono essere specificati per richiedere che l'operatore aggregato sia applicato rispettivamente a tutti i valori o a tutti i valori **distinti** dell'espressione. ALL è il default, e in MySQL non si deve specificare.
- › Gli operatori aggregati scartano (non usano nei calcoli) i valori null. L'unica eccezione è costituita dalla funzione COUNT(*), che restituisce il conteggio totale dei record indipendentemente dal loro contenuto.
 - Su una tabella vuota tutti gli operatori di gruppo hanno valore null tranne COUNT, che vale zero.
- › Gli operatori aggregati possono anche apparire in espressioni aritmetiche complesse.
- › **Se è presente un operatore aggregato nella clausola SELECT, SQL non accetta che vengano specificate altre colonne** (o espressioni legate a colonne) che non siano a loro volta calcolate con operatori aggregati. Infatti, in questo caso si mescolerebbero valori derivati da singoli record a valori calcolati su record aggregati, creando ambiguità.
 - In questi casi, la soluzione è l'aggregazione esplicita dei record, come vedremo.



Esempi

```
//Calcolare il numero di valori distinti dell'attributo stipendio fra tutte le righe di impiegato
```

```
SELECT COUNT(DISTINCT stipendio) FROM impiegati
```

```
//numero di date di assunzione non nulle tra gli impiegati
```

```
SELECT COUNT(ALL data_assunzione) FROM impiegati
```

```
//numero di date di assunzione DISTINTE tra gli impiegati
```

```
SELECT COUNT(DISTINCT data_assunzione) FROM impiegati
```

```
//Determinare il numero di impiegati del reparto 1
```

```
SELECT COUNT(*) FROM impiegati WHERE IDreparto=1
```

```
//quanti impiegati lavorano a Roma?
```

```
SELECT COUNT(*) FROM impiegati JOIN reparti ON (impiegati.IDreparto = reparti.ID)  
WHERE reparti.citta='Roma'
```

```
//Calcolare la somma degli stipendi complessivi del reparto Amministrazione
```

```
SELECT SUM(stipendio+premio) FROM impiegati, reparti WHERE  
reparti.nome='Amministrazione' AND impiegati.IDreparto=reparti.ID
```

```
//Determinare il massimo stipendio tra gli impiegati che lavorano in un reparto di Roma
```

```
SELECT MAX(stipendio) FROM impiegati, reparti WHERE impiegati.IDreparto=reparti.ID  
AND reparti.citta='Roma'
```

```
//Calcolare lo stipendio massimo, minimo e medio tra gli impiegati
```

```
SELECT MAX(stipendio), AVG(stipendio), MIN(stipendio) FROM impiegati
```



Raggruppamento di Record

- › Finora abbiamo usato gli operatori aggregati per calcolare funzioni su tutti i record selezionati da una query, raggruppati in un unico «blocco». Spesso è invece necessario **calcolare funzioni aggregate su sotto-gruppi di record**.
 - Ad esempio per «fornire la somma degli stipendi degli impiegati **in ciascun reparto**», una query del tipo `SELECT IDreparto, SUM(stipendio) FROM impiegato` verrebbe respinta in quanto nella `SELECT` sono presenti operatori aggregati e espressioni non aggregate
- › Per ottenere il risultato desiderato, dobbiamo prima forzare un raggruppamento di record tramite la clausola `GROUP BY`, posta alla fine della query ma prima della clausola `ORDER BY`:
`SELECT ... FROM ... [WHERE ...] [GROUP BY ...] [ORDER BY ...]`
- › Dal punto di vista del DBMS, la query con questa nuova clausola viene eseguita secondo la seguente procedura:
 1. Si genera il **prodotto cartesiano** di tutte le tabelle specificate dalla clausola `FROM`, nell'ordine dato.
 2. Si applica la **condizione** di ricerca specificata nella clausola `WHERE` ai record generati al passo precedente.
 3. Si **partizionano i record** filtrati ottenuti al passo precedente in base al valore di una o più colonne, come specificato dalla clausola `GROUP BY`.
 4. Si creano tanti **record di output** quante sono le partizioni ottenute, con i valori associati calcolati dalle espressioni inserite nella clausola `SELECT`.
 5. Si **ordinano** i record di output usando, nell'ordine, i criteri di ordinamento specificati con la clausola `ORDER BY`.



Raggruppamento di Record

Clausola GROUP BY

- › La clausola GROUP BY permette di definire i criteri di partizionamento dei record in sotto-gruppi, su cui lavoreranno gli operatori aggregati:
GROUP BY *nome_colonna*, *nome_colonna*, ...
 - *nome_colonna* è il nome di una colonna valida nella tabella generata dalla clausola FROM
- › Se si specificano più colonne di raggruppamento, **le partizioni ottenute tramite i valori della prima colonna verranno sotto-partizionate usando i valori della seconda colonna** e così via. Le partizioni finali saranno quelle ottenute da tutti i livelli di partizionamento in sequenza.
 - I record con valori null in una colonna di raggruppamento vengono posti nello stesso gruppo.
- › Quando si specifica una GROUP BY, **nelle clausole SELECT e ORDER BY** possono essere specificati operatori aggregati, calcolati su ciascuna delle partizioni, ma anche **colonne o espressioni su colonne utilizzate nella GROUP BY** (che in questo caso «caratterizzano» correttamente la partizione da cui deriva il record).
 - In SQL:99 è possibile specificare colonne non inserite nella GROUP BY ma funzionalmente dipendenti da esse. Tuttavia, questa pratica è opzionale.



Esempi

```
//Calcolare la somma degli stipendi degli impiegati in ciascun reparto
SELECT IDreparto, SUM(stipendio) FROM impiegati GROUP BY IDreparto
```

```
//Determinare il massimo stipendio in ogni reparto, ordinando i risultati
SELECT IDreparto, MAX(stipendio) AS stip_massimo
FROM impiegati
GROUP BY IDreparto
ORDER BY stip_massimo
```

```
//Determinare la somma degli stipendi dei vari reparti (di cui vogliamo conoscere il
nome)
SELECT reparti.nome, SUM(impiegati.stipendio)
FROM impiegati, reparti
WHERE impiegati.IDreparto = reparti.ID
GROUP BY reparti.ID
//ERRATA! Tutte le colonne non risultanti di una operazione di aggregazione presenti
//nella clausola SELECT devono essere anche presenti in quella GROUP BY!
```

```
//Le due seguenti interrogazioni sono invece corrette:
SELECT reparti.nome, SUM(impiegati.stipendio)
FROM impiegati, reparti
WHERE impiegati.IDreparto = reparti.ID
GROUP BY reparti.nome
```

```
SELECT reparti.nome, SUM(impiegati.stipendio) FROM impiegati, reparti
WHERE impiegati.IDreparto = reparti.ID
GROUP BY reparti.ID, reparti.nome
```

```
//Supponiamo di voler raggruppare gli impiegati sulla base del reparto e della mansione;
per ogni gruppo si vogliono determinare il nome del reparto e della mansione
corrispondenti, quanti impiegati appartengono al gruppo la somma e la media dei relativi
stipendi
SELECT reparti.nome, mansione, count(*), SUM(stipendio), AVG(stipendio)
FROM impiegati JOIN reparti ON (reparti.ID= impiegati.IDreparto)
GROUP BY reparti.nome, mansione
```



Raggruppamento di Record: Condizioni di Gruppo

- › Come abbiamo visto, la clausola WHERE filtra i record prima che vengano raggruppati come richiesto dalla GROUP BY. Non è possibile quindi porre condizioni sui valori ottenuti dal raggruppamento stesso
 - Ad esempio per «fornire la somma degli stipendi degli impiegati in ciascun reparto quando questa supera la soglia X»
- › A questo scopo, SQL mette a disposizione una clausola che lavora come la WHERE, ma sui valori di operatori aggregati, e permette quindi di specificare condizioni **sui gruppi di record** generati dalla GROUP BY: la clausola **HAVING**:
SELECT ... FROM ... [WHERE ...] [GROUP BY ...] [HAVING ...] [ORDER BY ...]
- › Dal punto di vista del DBMS, la query con questa nuova clausola viene eseguita secondo la seguente procedura:
 1. Si genera il **prodotto cartesiano** di tutte le tabelle specificate dalla clausola FROM, nell'ordine dato.
 2. Si applica la **condizione** di ricerca specificata nella clausola WHERE ai record generati al passo precedente.
 3. Si **partizionano i record** filtrati ottenuti al passo precedente in base al valore di una o più colonne, come specificato dalla clausola GROUP BY.
 4. Se **selezionano le partizioni** che rendono vera la condizione espressa con la clausola HAVING.
 5. Si creano tanti **record di output** quante sono le partizioni ottenute, con i valori associati calcolati dalle espressioni inserite nella clausola SELECT.
 6. Si **ordinano** i record di output usando, nell'ordine, i criteri di ordinamento specificati con la clausola ORDER BY.



Raggruppamento di Record: Condizioni di Gruppo

Clausola HAVING

- › La clausola HAVING permette di specificare un'espressione di filtro che verrà applicata ai singoli sotto-gruppi generati dalla clausola GROUP BY:
HAVING *espressione*
 - Dove *espressione* ha la stessa sintassi delle clausole WHERE, però non può utilizzare direttamente le colonne della tabella, **ma solo solo operatori aggregati applicati ad esse.**
- › Solo i sotto-gruppi per cui l'espressione HAVING vale true saranno passati alla SELECT per la generazione dell'output.



Esempi

//Supponiamo di voler raggruppare gli impiegati sulla base del reparto e della mansione; per ogni gruppo si vogliono determinare il nome del reparto e della mansione corrispondenti, quanti impiegati appartengono al gruppo la somma e la media dei relativi stipendi. Siamo interessati solo ai gruppi che contengono almeno due impiegati.

```
SELECT reparti.nome, mansione, count(*), SUM(stipendio), AVG(stipendio)
FROM impiegati JOIN reparti ON (reparti.ID= impiegati.IDreparto)
GROUP BY reparti.nome, mansione
HAVING count(*) >= 2;
```

//Determinare i reparti che spendono più di 100 in stipendi

```
SELECT IDreparto, SUM(stipendio) AS somma_stipendi
FROM impiegati
GROUP BY IDreparto
HAVING SUM(stipendio) > 100
```

//Determinare i reparti in cui la media degli stipendi dei dirigenti è superiore a 25

```
SELECT IDreparto
FROM impiegati
WHERE mansione='dirigente'
GROUP BY IDreparto
HAVING AVG(stipendio) > 25
```




Limitazione dell'output

- › Molti DBMS, tra cui MySQL, prevedono un'ultima clausola **LIMIT** utile a limitare il numero di righe ritornate da una query di selezione.
SELECT ... FROM ... [WHERE ...] [GROUP BY ...] [HAVING ...] [ORDER BY ...] [LIMIT ...]
- › Questo può essere utile per evitare di far estrarre e trasmettere al client troppi record quando ne sono necessari solo alcuni, ad esempio:
 - per visualizzare solo i primi n record in un certo ordine (ORDER BY)
 - per paginare i record (quindi estrarne solo una pagina alla volta)
- › In MySQL, la clausola LIMIT ha la seguente sintassi:
LIMIT [offset,] lunghezza
 - *offset*, se specificato, è l'indice del primo record da estrarre (base zero). Se omissso, l'offset vale zero.
 - *lunghezza* indica il numero di record da estrarre
- › Dal punto di vista del DBMS, la query con questa nuova clausola viene eseguita secondo la seguente procedura:
 1. Si genera il **prodotto cartesiano** di tutte le tabelle specificate dalla clausola FROM, nell'ordine dato.
 2. Si applica la **condizione** di ricerca specificata nella clausola WHERE ai record generati al passo precedente.
 3. Si **partizionano i record** filtrati ottenuti al passo precedente in base al valore di una o più colonne, come specificato dalla clausola GROUP BY.
 4. Se **selezionano le partizioni** che rendono vera la condizione espressa con la clausola HAVING.
 5. Si creano tanti **record di output** quante sono le partizioni ottenute, con i valori associati calcolati dalle espressioni inserite nella clausola SELECT.
 6. Si **ordinano** i record di output usando, nell'ordine, i criteri di ordinamento specificati con la clausola ORDER BY.
 7. Si **limita** l'output ai soli record specificati dalla clausola LIMIT



Esempi

```
//Determinare l'impiegato con lo stipendio più alto
```

```
SELECT cognome
```

```
FROM impiegati
```

```
ORDER BY stipendio DESC
```

```
LIMIT 1
```

//ATTENZIONE: se più impiegati sono "primi a pari merito", questa interrogazione ne restituisce solo uno!

```
//Estrarre i primi dieci impiegati in ordine alfabetico
```

```
SELECT nome, cognome
```

```
FROM impiegati
```

```
ORDER BY cognome, nome
```

```
LIMIT 10
```



Subquery

- › Una delle ragioni che rendono SQL un linguaggio potente è la possibilità di esprimere query complesse in termini di query più semplici, tramite il meccanismo delle subquery.
- › Le clausole SELECT e WHERE di una query (detta query esterna) possono infatti contenere altre query (dette subquery).
 - Le subquery vengono usate per estrarre uno o più valori da utilizzare in un predicato (clausola WHERE) o da estrarre (clausola SELECT) della query esterna .
 - È anche possibile per una subquery avere al suo interno altre subquery.
- › Una subquery può fare riferimento **ai campi del record corrente** (cioè quello sul quale sta venendo valutata) **delle query esterne**.
- › **I DBMS possono imporre restrizioni sulla complessità delle subquery!**



Subquery scalari

- › Le **query scalari**, cioè che ritornano **un solo record costituito da una sola colonna** (anche calcolata), possono essere usate come **argomenti degli operatori di confronto nella clausola WHERE**:
WHERE *espressione* = (SELECT x FROM y...)
- › Le query scalari possono essere inoltre usate come espressioni che ritornano valori **nella clausola SELECT**:
SELECT a,(SELECT x FROM y...),b FROM z...
- › Se la subquery restituisce più di un record, SQL restituisce un errore.



Subquery riga

- › Le **query riga** (row subquery), cioè che restituiscono **un solo record composto da più colonne**, possono essere utilizzate nelle clausole WHERE con operatori di confronto usando una speciale sintassi:
WHERE (*espressione1*,*espressione2*) = SELECT (x1,x2 FROM y...)
 - In MySQL, la lista di espressioni da confrontare deve essere preceduta dalla parola chiave **ROW**:
WHERE ROW(*espressione1*,*espressione2*) = SELECT (x1,x2 FROM y...)
- › Gli operatori di confronto diversi da '=' si comportano in modo particolare con le row subquery. Ad esempio, $(a, b) < (x, y)$ equivale a $(a < x)$ OR $((a = x) \text{ AND } (b < y))$
- › Se la subquery restituisce più di un record, SQL restituisce un errore.



Subquery lista (o colonna)

- › Le **query lista (o query colonna)**, cioè che ritornano più record costituiti da una sola colonna, possono essere usate
 - come secondo argomento **dell'operatore IN**:
WHERE *espressione* [NOT] IN (subquery)
 - › l'operatore [NOT] IN vale true se l'*espressione* (non) ha un valore tra quelli estratti dalla *subquery*.
 - con gli operatori di confronto, facendoli precedere dalle parole chiave **ALL** o **ANY**:
WHERE *espressione* = ANY | ALL (subquery)
 - › SQL esegue il confronto richiesto tra il valore dell'*espressione* e ciascuno dei valori estratti dalla *subquery*. Utilizzando ANY, questi confronti vengono messi in OR (*il confronto globale è vero se vale per almeno una coppia di valori*), mentre con ALL vengono messi in AND (*il confronto globale è vero se vale per tutte le coppie di valori*).



Subquery generiche nell'operatore EXIST

- › Ogni **query generica** può essere infine utilizzata come subquery utilizzandola come argomento dell'operatore **EXIST**
WHERE [NOT] EXIST(subquery)
 - L'espressione è true (false) se la subquery (non) restituisce almeno un record.



Esempi: subquery scalari

```
//Determinare gli impiegati con lo stipendio più alto
SELECT cognome
FROM impiegati
WHERE stipendio = ( SELECT MAX(stipendio) FROM impiegati );
```

```
//Si vogliono elencare tutti gli impiegati che hanno la stessa mansione
dell'impiegato Rossi
SELECT cognome
FROM impiegati
WHERE mansione = (SELECT mansione FROM impiegati WHERE cognome = 'Rossi')
//la subquery restituisce come valore "dirigente"; la query esterna determina
quindi tutti gli impiegati che sono dirigenti (Rossi incluso).
```

```
//la stessa interrogazione, però, si può ottenere con una singola query
usando opportunamente dei join:
SELECT altro.cognome
FROM impiegati rossi, impiegati altro
WHERE rossi.mansione = altro.mansione AND rossi.cognome = 'Rossi'
```

```
//Si vogliono elencare tutti gli impiegati che hanno uno stipendio superiore
alla media
SELECT cognome, stipendio
FROM impiegati
WHERE stipendio > ( SELECT AVG(stipendio) FROM impiegati);
```




Esempi: subquery riga e lista

```
//Elencare gli impiegati con la stessa mansione e stipendio di Rossi
SELECT cognome
FROM impiegati
WHERE (mansione,stipendio) = ( SELECT mansione,stipendio FROM impiegati WHERE
cognome = 'Rossi');
```



```
//Tutti i dati dei reparti in cui sono stati assunti nuovi impiegati negli ultimi
30 giorni
SELECT *
FROM reparti
WHERE ID IN (SELECT IDreparto FROM impiegati WHERE data_assunzione >
DATE_SUB(NOW(), INTERVAL 30 DAY))
```



```
//versione senza subquery (più complessa da interpretare!)
SELECT DISTINCT reparti.*
FROM reparti JOIN impiegati ON (reparti.ID = impiegati.IDreparto)
WHERE impiegati.data_assunzione > DATE_SUB(NOW(), INTERVAL 30 DAY)
```



```
//Tutti gli impiegati che lavorano in un reparto di Roma
SELECT *
FROM impiegati
WHERE impiegati.IDreparto = ANY (SELECT ID FROM reparti WHERE citta='Roma')
```



```
//Gli impiegati che guadagnano più di ogni college di Roma
SELECT *
FROM impiegati
WHERE impiegati.stipendio > ALL ( SELECT stipendio FROM impiegati JOIN reparti ON
(impiegati.IDreparto=reparti.ID) WHERE citta='Roma' )
```



Subquery correlate

- › In una query nidificata è possibile fare riferimento alle tabelle (o agli alias delle stesse) definite nella clausola FROM della query esterna.
- › L'uso di alias sulle tabelle in questi casi è sempre consigliabile per evitare ambiguità.
 - Se una query nidificata importa una tabella con lo stesso nome (o alias) della sua query esterna, la tabella più interna nasconde quella esterna. In tal caso vanno utilizzati alias diversi.
- › In questo modo, si possono correlare le due query, cioè usare i valori del record attualmente in fase di valutazione da parte della WHERE esterna all'interno della query nidificata.



Esempi

```
//Elencare tutti gli impiegati che hanno uno stipendio minore di altri impiegati
assunti nella stessa data
//versione singola query
SELECT i1.*
FROM impiegati i1, impiegati i2
WHERE i1.data_assunzione=i2.data_assunzione AND i2.stipendio>i1.stipendio
//versione con subquery e quantificatore
SELECT i1.*
FROM impiegati i1
WHERE i1.stipendio < ANY (SELECT stipendio FROM impiegati i2 WHERE
i1.data_assunzione=i2.data_assunzione )
//versione con subquery e predicato EXISTS
SELECT i1.*
FROM impiegati i1
WHERE EXISTS (SELECT * FROM impiegati i2 WHERE
i1.data_assunzione=i2.data_assunzione AND i2.stipendio>i1.stipendio )

//Elencare tutti gli impiegati che hanno uno stipendio minore di TUTTI GLI ALTRI
impiegati assunti nella stessa data
//versione con subquery e quantificatore
SELECT i1.*
FROM impiegati i1
WHERE i1.stipendio < ALL (SELECT stipendio FROM impiegati i2 WHERE
i1.data_assunzione=i2.data_assunzione )
//versione con subquery e predicato EXISTS
SELECT i1.*
FROM impiegati i1
WHERE NOT EXISTS (SELECT * FROM impiegati i2 WHERE
i1.data_assunzione=i2.data_assunzione AND i2.stipendio<=i1.stipendio )
```



Query di unione, intersezione e differenza

- › SQL permette di effettuare operazioni insiemistiche tra gli insiemi di record restituiti da due query: unione, intersezione, differenza
query1 UNION query2
query 1 MINUS query2
query 1 INTERSECT query2
- › Perché questo possa avvenire, **le query coinvolte devono presentare lo stesso numero di colonne, con tipi compatibili** (e a volte lo stesso nome)
- › MySQL supporta solo le unioni: gli altri tipi di operazione sono ottenibili tramite normali query, con una definizione più complessa.
- › In MySQL, l'unione avrà come nomi di colonna i nomi derivanti dalla *query1*
- › Per default, MySQL effettua una **UNION DISTINCT**, eliminando le righe duplicate durante l'unione. Se si vuole eliminare questo effetto, si può indicare esplicitamente **UNION ALL**
- › E' possibile porre tra parentesi le definizioni delle due query argomento dell'unione per evitare ambiguità
- › E' possibile inserire una clausola **LIMIT** e/o **ORDER BY** che valga sull'intera unione, ponendole alla fine della query composta e ricordando di mettere ciascuna query tra parentesi



Esempi

//vogliamo estrarre gli impiegati che hanno lo stipendio più alto e quelli che hanno il più basso

```
(
  SELECT *
  FROM impiegati
  WHERE stipendio = (SELECT MAX(stipendio) FROM impiegati)
)
UNION
(
  SELECT *
  FROM impiegati
  WHERE stipendio = (SELECT MIN(stipendio) FROM impiegati)
)
```



Viste

- › Le viste sono stored queries, cioè interrogazioni a cui viene assegnato un nome e la cui definizione è immagazzinata nel database:
CREATE VIEW *nome_vista* AS *query_di_selezione*
 - Dove *query_di_selezione* è una SELECT arbitraria. Sono possibili alcune limitazioni sulla complessità della query, che dipendono dal DBMS in uso.
 - **La definizione della vista è «congelata» al momento della creazione:** se una vista è definita come SELECT * in una tabella, nuove colonne aggiunte alla tabella non diventano parte della vista e colonne eliminate dalla tabella genereranno un errore quando si esegue la vista.
- › Le viste generano tabelle virtuali a tutti gli effetti, e come tali **utilizzabili all'interno delle clausola FROM di altre istruzioni SELECT** (comprese quelle di altre viste)
- › Per aggiornare la definizione di una vista o eliminarla, sono disponibili i corrispondenti comandi **ALTER VIEW** e **DROP VIEW**



Esempi

```
CREATE VIEW impiegati_anagrafica AS
SELECT CONCAT(nome, ' ', cognome) AS nome, codice_fiscale
FROM impiegati;

SELECT * FROM impiegati_anagrafica;

SELECT * FROM impiegati_anagrafica WHERE nome LIKE '%Carlo%'
```

SQL: Linguaggio Procedurale

Programmare procedure, funzioni e trigger direttamente del database





Procedure, Funzioni e Trigger

- › Procedure e trigger permettono di scrivere codice per implementare funzionalità e controlli sui dati direttamente nella base di dati.
- › In questa maniera, è possibile fornire agli utenti della base di dati un ulteriore strato di astrazione verso la struttura logica dei dati e assicurarsi che alcuni controlli essenziali siano sempre eseguiti e non delegati al programma client che si interfaccia con il DBMS.
- › Ad esempio, grazie all'uso delle procedure è possibile
 - evitare che un programmatore implementi scorrettamente le operazioni di aggiornamento dei dati (soprattutto nel caso coinvolgano più tabelle)
 - migliorare le prestazioni del database, perchè l'interazione tra il programma client e la base di dati diminuisce sensibilmente se delle operazioni complesse vengono gestite completamente dal DBMS



Procedure e Funzioni

- › Una procedura è un frammento di codice composto da istruzioni SQL dichiarative e procedurali memorizzate nella base di dati. Questo codice può essere attivato richiamandolo da un programma, da un'altra procedura o da un trigger, e può avere parametri di input (argomenti) e output (valori di ritorno).
- › Una funzione è una procedura che restituisce un valore. Alcuni DBMS non supportano le funzioni, oppure definiscono procedure e funzioni allo stesso modo.
- › Una procedura/funzione può essere composta da due tipi di istruzioni:
 - dichiarative: le istruzioni SQL viste finora (CREATE, UPDATE, SELECT), con speciali estensioni per gestire la memorizzazione dei valori estratti dalla base di dati in variabili;
 - procedurali: costrutti comuni dei linguaggi di programmazione, come IF-THEN-ELSE e WHILE-DO.
- › La sintassi delle procedure/funzioni non è oggetto di standardizzazione, per cui qui è possibile darne solo una versione generica, tipicamente accettata con piccole differenze da molti DBMS.
- › **Poiché il punto e virgola è usato da quasi tutti i DBMS come terminatore delle query, il suo uso all'interno delle procedure come separatore di istruzioni può produrre risultati inattesi.** A questo scopo, prima di dichiarare una procedura si può modificare temporaneamente il terminatore di query e poi ripristinarlo alla fine della definizione:
SET TERM *terminatore*
 - *terminatore* è una sequenza di caratteri non contenente spazi
 - In MySQL, si usa la sintassi **DELIMITER *terminatore***



Procedure e Funzioni in MySQL

Creazione

- › Per creare una procedura, si utilizza il seguente comando SQL:
CREATE PROCEDURE *nome_procedura* ([[IN|OUT|INOUT] *nome_parametro tipo_parametro*, ...]) *corpo*
 - *nome_procedura* è il nome univoco della procedura
 - l'elenco dei parametri della procedura, opzionale, è composto da coppie *nome_parametro tipo_parametro*, dove *nome_parametro* è un identificatore e *tipo_parametro* è uno dei tipi definiti da SQL. Opzionalmente, ogni parametro può essere preceduto dalle parole chiave
 - › **IN**: il parametro è di input, passato per valore
 - › **INOUT**: il parametro è di input e output, passato per riferimento
 - › **OUT**: il parametro è di output, inizialmente vale null ma il valore assegnatogli dalla procedura viene propagato al chiamante
 - Il *corpo* è costituito da istruzioni procedurali valide, solitamente poste all'interno di un blocco delimitato dalle parole chiave BEGIN e END. Le singole istruzioni sono separate tra loro da un punto e virgola.
- › Per eliminare una procedura preesistente, basta invocare il comando
DROP PROCEDURE *nome_procedura*



Procedure e Funzioni in MySQL

Creazione

- › Per creare una funzione, si utilizza un comando SQL molto simile:
CREATE FUNCTION *nome_funzione* ([*nome_parametro tipo_parametro*, ...])
RETURNS *tipo_risultato corpo*
 - *nome_funzione* è il nome univoco della funzione
 - l'elenco dei parametri della, opzionale, è composto da coppie *nome_parametro tipo_parametro*, dove *nome_parametro* è un identificatore e *tipo_parametro* è uno dei tipi definiti da SQL. **Le funzioni non possono avere parametri di output.**
 - *tipo_risultato* è uno dei tipi definiti da SQL.
 - Il *corpo* è costituito da istruzioni procedurali valide, solitamente poste all'interno di un blocco delimitato dalle parole chiave BEGIN e END. Le singole istruzioni sono separate tra loro da un punto e virgola.
 - E' possibile terminare una funzione in qualunque momento restituendo il suo valore di ritorno con l'istruzione
RETURN espressione
- › In MySQL, **le funzioni devono essere sempre deterministiche**, cioè ritornare sempre lo stesso risultato se eseguite sugli stessi dati e con gli stessi parametri. E' necessario dichiarare questa condizione inserendo la parola chiave **DETERMINISTIC** dopo il *tipo_risultato*.
- › Per eliminare una funzione preesistente, basta invocare il comando
DROP FUNCTION *nome_funzione*



Istruzioni nelle Procedure/Funzioni

Blocchi e variabili

- › Il corpo di una procedura o funzione è costituito da un **blocco** di istruzioni delimitato da **BEGIN** ed **END**.
 - Ogni blocco può nidificare altri blocchi
 - Se il blocco è costituito da una singola istruzione, si possono omettere BEGIN ed END.
- › I blocchi, e anche le istruzioni di loop che vedremo più avanti, possono essere **etichettati**:
label: BEGIN ... END
 - Per uscire immediatamente da un blocco etichettato con una label (particolarmente utile nel caso dei loop), si può scrivere
LEAVE label
- › Un blocco può iniziare con una o più dichiarazioni di **variabili locali**, che potranno essere poi usate all'interno del codice della procedura stessa. La sintassi della dichiarazione è la seguente:
DECLARE nome_variabile tipo_variabile
 - dove *nome_variabile* è un identificatore e *tipo_variabile* è uno dei tipi definiti da SQL.
- › Successivamente, per impostare il valore di una variabile si può usare l'istruzione SET:
SET nome_variabile = espressione



Invocare una procedura o una funzione

- › Una procedura può essere chiamata usando la seguente sintassi:

CALL *nome_procedura* [(*argomento1*, ...)]

- Se la procedura costituisce una query parametrica, la sua chiamata corrisponderà all'esecuzione della query con i parametri dati.
- Se la procedura contiene parametri di output (OUT), al posto del corrispondente argomento dovrà essere inserita una variabile precedentemente dichiarata, se la chiamata viene fatta dall'interno di un'altra procedura/funzione o, se la chiamata avviene direttamente dall'interprete SQL, una variabile globale creata con la speciale sintassi **@nome**, che potrà poi essere utilizzata in altri statement.

- › Una funzione, che ritorna quindi un valore, può essere invece usata in qualsiasi espressione SQL valida, ad esempio all'interno di una SELECT o di una WHERE:

Ad esempio, **SELECT *nome_funzione*([*argomento1*,...])**



Esempi

```
CALL incrementa_stipendi();
```

```
CALL impiegati_per_lettera('r');
```

```
SELECT livello_stipendiale_minimo();
```

```
CALL funzione_con_parametro_out(123,@var);
```

```
SELECT @var;
```



Query all'interno di Procedure

- › Le normali query SQL presenti all'interno del corpo di una procedura vengono eseguite come di consueto.
- › Una query può essere presente in una procedura in varie forme:
 - Per le **query scalari**, come parte di un'espressione
 - Per le **query riga**, nella speciale forma SELECT...INTO
 - Per **qualsiasi tipo di query**, nella forma diretta o gestita tramite un cursore



Query all'interno di Procedure

Query generali / dirette

- › Le query di forma generale (quindi non limitate a query scalari o riga), possono essere inserite in una procedura in maniera diretta, come una qualsiasi istruzione. In questo caso la query viene eseguita normalmente.
- › E' possibile anche scrivere procedure il cui corpo è costituito da una singola query.
- › **Il risultato dell'ultima query eseguita in questa modalità viene restituito come output della chiamata a procedura.**
- › Ovviamente la query può usare al suo interno variabili e parametri della procedura/funzione: in questo modo si possono usare le procedure per realizzare e inserire nella base di dati delle **query parametriche**.
- › E' possibile usare funzioni utili dell'SQL quali **found_rows()** (che restituisce il numero di record estratti dall'ultima SELECT) e **last_insert_id()** (che restituisce l'ID dell'ultimo record inserito) per verificare i risultati dello statement SQL.



Query all'interno di Procedure

Query scalari

- › Le query scalari possono essere usate ovunque il linguaggio procedurale accetti un'espressione che genera un valore.
- › La query scalare, a tutti gli effetti, viene valutata come il valore (scalare) che essa restituisce.
- › Ad esempio si possono trovare espressioni del tipo
 - **SET** variabile = (query);
 - **RETURN** (query);
- › Ovviamente la query può usare al suo interno variabili e parametri della procedura/funzione.
- › *Se la query restituisce più righe o più colonne, verrà generato un errore.*



Query all'interno di Procedure

Query riga

- › Le query riga possono essere invocate, assegnando i valori di tutte le colonne restituite a una serie di variabili, utilizzando la speciale forma **SELECT x,y ... INTO *variabile1, variabile2*,...**
 - Le variabili devono essere dello stesso numero e avere tipi compatibili con le colonne estratte dalla SELECT
- › Ovviamente la query può usare al suo interno variabili e parametri della procedura/funzione.
- › *Se la query restituisce più righe o più colonne di quelle assegnate verrà generato un errore.*



Esempi

```
DELIMITER $$

//selezioniamo le righe della tabella impiegati corrispondenti agli impiegati
//il cui cognome inizia per una determinata lettera
CREATE PROCEDURE impiegati_per_lettera(lettera char(1))
    SELECT * FROM impiegati WHERE cognome LIKE CONCAT(lettera,'%')$$

//incrementiamo (fino alla media) gli stipendi al di sotto della media
CREATE PROCEDURE incrementa_stipendi()
BEGIN
    DECLARE media DECIMAL(7,2);
    SELECT AVG(stipendio) FROM impiegati INTO media;
    UPDATE impiegati SET stipendio=media WHERE stipendio<media;
END$$

//possiamo anche modularizzare incrementa_stipendi usando una funzione:
CREATE FUNCTION livello_stipendiale_minimo() RETURNS DECIMAL(7,2)
BEGIN
    RETURN (SELECT AVG(stipendio) FROM impiegati);
END$$
DROP PROCEDURE incrementa_stipendi$$
CREATE PROCEDURE incrementa_stipendi()
BEGIN
    DECLARE media DECIMAL(7,2);
    SET media = livello_stipendiale_minimo();
    UPDATE impiegati SET stipendio=media WHERE stipendio<media;
END$$

DELIMITER ;
```



Istruzioni nelle Procedure/Funzioni

Costrutti di flusso / condizionali

- › Il primo costrutto condizionale messo a disposizione da SQL è il classico IF-THEN-ELSE:

IF *condizione1* THEN *corpo1* [ELSEIF *condizione2* THEN *corpo2* ...] [ELSE *corpo3*] END IF

- Notare che è possibile scrivere varie condizioni alternative usando istruzioni **ELSEIF**

- › Costrutti condizionali più complessi sono quelli ottenibili con la parola chiave CASE:

CASE *espressione* WHEN *espressione_caso1* THEN *corpo_caso1* ... [ELSE *corpo_else*] END CASE

- In questo caso, viene eseguito il *corpo_casoN* della prima WHEN la cui *espressione_casoN* ha lo stesso valore dell'*espressione* di controllo.
- Si tratta della stessa semantica del noto *switch* di Java, ma senza la parola chiave *break*.

- › **CASE WHEN *espressione_caso1* THEN *corpo_caso1* ... [ELSE *corpo_else*] END CASE**

- In questo secondo caso, viene eseguito il *corpo_casoN* della prima WHEN la cui *espressione_casoN* ha valore *true*

- › In entrambi i casi, se nessuna delle WHEN si attiva, viene seguito il *corpo_else* se presente, altrimenti **viene generato un errore**.



Istruzioni nelle Procedure/Funzioni

Costrutti di flusso / loop

- › Ci sono tre istruzioni di loop. Tutte possono essere opzionalmente etichettate con una *label*:
 - **[*label*:] LOOP corpo END LOOP**
E' un loop infinito. L'unico modo di uscirne è usare la parola chiave **LEAVE** (se il loop ha un'etichetta).
 - **[*label*:] REPEAT corpo UNTIL condizione END REPEAT**
E' il classico loop REPEAT-UNTIL.
 - **[*label*:] WHILE condizione DO corpo END WHILE**
E' il classico loop WHILE.
- › In tutti i loop con etichetta,
 - per saltare immediatamente all'iterazione successiva, si può usare il comando **ITERATE *label***
 - Per uscire immediatamente dal loop, si può scrivere **LEAVE *label***



Esempi

```
//livello_stipendiale_minimo con possibilità di specificare un reparto
specifico
CREATE FUNCTION livello_stipendiale_minimo(IDreparto INTEGER) RETURNS
DECIMAL(7,2)
BEGIN
    /* nell'espressione IF possiamo usare tutti gli operatori visti per l'SQL
    */
    IF IDreparto IS NULL THEN
        /* la query va messa tra parentesi per evitare problemi di sintassi */
        RETURN (SELECT AVG(stipendio) FROM impiegati);
    ELSE
        /* notare l'uso del nome della tabella per disambiguare il parametro
        omonimo */
        RETURN (SELECT AVG(stipendio) FROM impiegati WHERE impiegati.IDreparto =
IDreparto);
    END IF;
END$$
DELIMITER ;

SELECT livello_stipendiale_minimo(null),livello_stipendiale_minimo(1);
```



Conditions

Dichiarazione

- › Le *conditions* sono segnalazioni di eventi specifici all'interno del corpo delle procedure che richiedono un trattamento speciale.
 - Le *conditions* possono essere viste come le eccezioni dell'SQL, anche se hanno un uso più generale (non segnalano solo condizioni di errore).
- › MySQL dispone di una serie di conditions predefinite.
- › Se necessario, è possibile **dichiarare le proprie condition** (in un parallelo con Java, è come se si dichiarasse una classe derivata da Exception) con la sintassi **DECLARE nome CONDITION FOR valore**
 - *Nome* è il nome da dare alla condizione, per invocarla o gestirla in seguito
 - *Valore* può essere un intero indicante un codice di errore predefinito o l'espressione **SQLSTATE numero** indicante uno stato SQL predefinito.
 - **Per segnalare un'eccezione generale definita dall'utente, usare l'SQLSTATE 45000**
- › Per **attivare una condizione**, si utilizza la sintassi **SIGNAL condizione [SET attributo=valore, ...]**
 - *Condizione* può essere un **SQLSTATE numero** oppure il nome di una condizione definita dall'utente.
 - È possibile valorizzare anche una serie di attributi (predefiniti) della condizione, come **MESSAGE_TEXT**.



Conditions

Handling

- › Le conditions devono essere catturate utilizzando degli appositi handler, che possono essere dichiarati in ciascun blocco, subito dopo la dichiarazione di variabili, condizioni e cursori:
DECLARE [CONTINUE | EXIT] HANDLER FOR *condizione* BEGIN *corpo_handler* END
 - In un parallelo con i linguaggi di programmazione, gli handler sono i blocchi *catch* di una istruzione *try*, che in questo caso corrisponde all'intero blocco BEGIN...END contenente l'handler stesso.
- › Quando viene attivata una condition, l'interprete SQL cerca e attiva il primo handler compatibile nel blocco corrente o, se non presente, in uno dei blocchi più esterni (in caso di BEGIN...END nidificati)
 - Il codice nel *corpo_handler* viene eseguito e può contenere qualsiasi istruzione, tranne **ITERATE** o **LEAVE**.
 - Il codice nel *corpo_handler* può risollevare la stessa condizione, eventualmente alterandone gli attributi, tramite l'istruzione **RESIGNAL**.
- › Dopo l'esecuzione del *corpo_handler* l'esecuzione
 - Continua dal punto del codice che aveva generato la condition, se viene specificato **CONTINUE**.
 - Continua dall'istruzione che segue la fine del blocco contenente l'handler, se viene specificato **EXIT**.



Query all'interno di Procedure

Query generali / cursori

- › Per le query di forma generale è inoltre possibile iterare sui risultati eseguendo operazioni su ciascuna riga restituita utilizzando i **cursori**.
 - Come vedremo meglio più avanti, i cursori sono un meccanismo di adattamento tra la programmazione procedurale (non solo quella di SQL, ma anche PHP, Java, C, ecc.), le cui strutture solitamente non permettono di gestire efficientemente intere tabelle ma solo singoli record, e le tabelle generate della query SQL.
- › Un cursore, all'atto pratico, è **un puntatore a uno dei record generati da una query**, che permette di **leggere le colonne del record corrente e può essere spostato avanti** (e a volte indietro).
- › Per usare un cursore, bisogna prima di tutto **dichiararlo** e associarlo alla query sui cui risultati dovrà iterare. Le dichiarazioni di cursore possono comparire in ogni blocco, subito dopo quelle delle variabili:
DECLARE nome_cursore CURSOR FOR istruzione_select
 - dove *nome_cursore* è l'identificatore del cursore da attivare e *istruzione_select* è una SELECT SQL standard, che può anche usare variabili/parametri nella sua definizione
- › La dichiarazione del cursore non esegue subito la query: quando si desidera utilizzarlo, infatti, è necessario **aprirlo**:
OPEN nome_cursore
- › Una volta aperto il cursore, possiamo scorrere in avanti tra i record dell'istruzione select associata, una riga per volta, scrivendo:
FETCH nome_cursore INTO variabile1, variabile2,...
 - Le variabili devono essere dello stesso numero e avere tipi compatibili con le colonne estratte dalla SELECT associata al cursore
 - Solitamente questa istruzione viene posta in un loop
- › Una volta terminato di leggere i dati della query, il cursore va chiuso:
CLOSE nome_cursore
 - I cursori dichiarati in un blocco (BEGIN...END) sono chiusi automaticamente alla sua uscita.



Query all'interno di Procedure

Query generali / cursori e condizioni

- › Quando un cursore viene spostato oltre l'ultima riga disponibile, SQL genera una condizione *No Data* (o *Not Found*) che va catturata usando un handler de tipo

```
DECLARE [CONTINUE | EXIT] HANDLER FOR NOT FOUND BEGIN  
corpo_handler END
```

- › Il codice nel *corpo_handler* può, ad esempio, impostare una variable che determina l'uscita dal loop di lettura del cursore (usando la modalità **CONTINUE**) oppure uscire direttamente dal blocco corrispondente (usando la modalità **EXIT**).



Esempi

```
DELIMITER $$
CREATE PROCEDURE gratifica_R_e_B()
BEGIN
    -- guardia per il loop del cursore
    DECLARE imp_data_available BOOLEAN;
    -- variabili in cui il cursore inserirà i dati
    DECLARE imp_cognome VARCHAR(50); DECLARE imp_id INTEGER;
    -- variabili di accumulo per statistiche
    DECLARE imp_with_R INTEGER; DECLARE imp_with_B INTEGER; DECLARE imp_other INTEGER;
    -- cursore che itera sulla query data
    DECLARE imp_crs CURSOR FOR SELECT ID, cognome FROM impiegati;
    -- handler per l'evento "fine record" all'interno del blocco corrente
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET imp_data_available = FALSE;
    -- inizializzazione
    SET imp_with_R=0; SET imp_with_B=0; SET imp_other=0; SET imp_data_available = TRUE;
    -- avvio query
    OPEN imp_crs;
    -- loop di lettura (etichettato)
    main: REPEAT
        -- lettura prossimo record
        FETCH imp_crs INTO imp_id, imp_cognome;
        -- verifichiamo se abbiamo letto qualcosa (cioè se l'handler non è scattato)
        IF imp_data_available THEN
            -- verifichiamo il cognome e aggiorniamo le statistiche
            CASE
                WHEN imp_cognome LIKE 'R%' THEN SET imp_with_R = imp_with_R+1;
                WHEN imp_cognome LIKE 'B%' THEN SET imp_with_B = imp_with_B+1;
                ELSE
                    SET imp_other = imp_other+1;
                    -- se il cognome non ci interessa, saltiamo il resto del loop
                    ITERATE main;
            END CASE;
            -- aumentiamo lo stipendio all'impiegato corrente
            UPDATE impiegati SET stipendio = stipendio+stipendio*0.25 WHERE ID=imp_id;
        END IF;
        -- chiudiamo il loop se non ci sono più dati
    UNTIL NOT imp_data_available END REPEAT;
    -- chiudiamo il cursore
    CLOSE imp_crs;
    -- TRUCCO: usiamo questa query a fine procedura per farci stampare le statistiche collezionate
    SELECT imp_with_R as numero_R_gratificati, imp_with_B as numero_B_gratificati, imp_other as non_gratificati;
END$$
DELIMITER ;
```

SQL: Trigger

Incorporare controlli avanzati e modifiche automatiche sui dati direttamente del database





Trigger

- › In molti DBMS è disponibile un costrutto detto *trigger* che esegue un blocco di codice sul database quando si verifica un *evento*.
 - Gli eventi gestibili mediante trigger sono **inserimenti, aggiornamenti e cancellazioni** su una tabella.
- › Dal punto di vista di SQL, i trigger sono molto simili alle procedure: si tratta di una sorta di procedura particolare che si attiva automaticamente in seguito a un determinato evento.
 - Il codice di un trigger deve essere semplice e di rapida esecuzione, in quanto viene eseguito dal DBMS a ogni manipolazione delle tabelle associate.
- › I trigger sono utilizzati per diversi scopi nella progettazione di un database:
 - mantenere l'integrità referenziale tra le varie tabelle
 - mantenere l'integrità dei dati in una singola tabella
 - calcolare automaticamente i valori di campi particolari ad ogni modifica dei dati ad essi correlati
- › La sintassi di un trigger, come quella di una procedura, non è completamente standardizzata.



Trigger in MySQL

- › Per creare un trigger, si usa il seguente comando SQL:
CREATE TRIGGER *nome_trigger* *tempo_trigger* *evento_trigger* ON *nome_tabella* FOR EACH ROW [*ordine_trigger*] *corpo_trigger*
 - *nome_trigger* è il nome univoco del trigger
 - *tempo_trigger* può essere **BEFORE** o **AFTER** e indica se il trigger deve essere eseguito *prima* o *dopo* che l'evento indicato sia stato gestito dal DBMS stesso
 - *evento_trigger* può essere **INSERT**, **UPDATE** o **DELETE** e indica l'operazione sulla tabella a cui è associato il trigger
 - *nome_tabella* identifica la tabella monitorata dal trigger
 - *ordine_trigger*, opzionale, può essere usato per ordinare i trigger per lo stesso evento sulla stessa tabella, scrivendo **FOLLOWS** o **PRECEDES** *altro_nome_trigger*
 - La parte **FOR EACH ROW** (*il trigger viene eseguito su ogni riga della tabella *nome_tabella* che subisce l'evento_trigger specificato*) è obbligatoria in MySQL, mentre altri DBMS supportano anche la modalità FOR EACH STATEMENT, che qui non tratteremo.
- › In generale, i trigger che devono poter bloccare l'operazione monitorata o modificare dei valori vanno attivati prima (BEFORE) che l'evento stesso sia gestito dal DBMS. Tutti gli altri trigger possono essere attivati dopo (AFTER) l'evento.
- › Per eliminare un trigger preesistente, basta invocare il comando
DROP TRIGGER *nome_trigger*



Manipolazione dei dati nei Trigger

- › All'interno dei trigger di tipo FOR EACH ROW è disponibile una speciale sintassi per riferirsi alla riga dati appena inserita/aggiornata/cancellata per la quale il trigger è stato attivato:
 - **NEW.nome_colonna**
serve a riferirsi al nuovo valore della colonna *nome_colonna* della riga appena aggiornata/inserita (se il trigger è AFTER UPDATE O INSERT) o da aggiornare/inserire (se il trigger è BEFORE UPDATE o INSERT)
 - **OLD.nome_colonna**
serve a riferirsi al valore precedente della colonna *nome_colonna* della riga appena aggiornata/cancellata (se il trigger è AFTER UPDATE O DELETE) o da aggiornare/cancellare (se il trigger è BEFORE UPDATE O DELETE)
- › Il corpo di un trigger può contenere qualsiasi istruzione ammessa per le procedure (comprese le dichiarazioni di variabili), ma non può restituire valori.
 - Sebbene nei trigger si possano usare istruzioni SQL come UPDATE, DELETE e INSERT, è necessario prestare molta attenzione alla possibilità che queste istruzioni attivino altri trigger "in cascata". Potrebbe infatti accadere che, in uno o più passi, gli eventi attivati dal codice del trigger richiamino il trigger di partenza, generando un ciclo infinito.
- › In particolare, nei trigger BEFORE, è possibile
 - manipolare direttamente le colonne del record NEW in modo da modificare il record da inserire o aggiornare. Questo permette al trigger di apportare correzioni o integrazioni ai dati proposti dall'utente.
SET NEW.nome_colonna = espressione
 - Segnalare una condizione di errore, causando la cancellazione dell'evento che ha scatenato il trigger (quindi l'inserimento/aggiornamento/cancellazione non avrà luogo)
SIGNAL SQLSTATE '45000' SET message_text = 'messaggio di errore'



Azioni dei Trigger

Cosa viene passato al trigger e cosa si può fare nel suo corpo

	BEFORE	AFTER
INSERT	<ul style="list-style-type: none">• (OLD non definito)• NEW il record che sta per essere inserito	<ul style="list-style-type: none">• (OLD non definito)• NEW il record appena inserito
UPDATE	<ul style="list-style-type: none">• OLD il record PRIMA dell'aggiornamento (stato corrente)• NEW il record DOPO l'aggiornamento (stato successivo)	<ul style="list-style-type: none">• OLD il record PRIMA dell'aggiornamento (stato precedente)• NEW il record DOPO l'aggiornamento (stato corrente)
DELETE	<ul style="list-style-type: none">• OLD il record da cancellare• (NEW non definito)	<ul style="list-style-type: none">• OLD il record cancellato• (NEW non definito)

	BEFORE	AFTER
INSERT	<ul style="list-style-type: none">• controllare la validità del record• sollevare eccezioni bloccando l'operazione• modificare il record prima dell'inserimento	<ul style="list-style-type: none">• modificare il resto del DB in base all'inserimento
UPDATE	<ul style="list-style-type: none">• controllare la validità del record• sollevare eccezioni bloccando l'operazione• modificare il record NEW prima dell'aggiornamento• annullare alcune modifiche reimpostando delle colonne di NEW ai valori di OLD	<ul style="list-style-type: none">• modificare il resto del DB in base all'aggiornamento



Esempi

```
DELIMITER $$
-- funzione di supporto
CREATE FUNCTION calcola_codice_fiscale(nome VARCHAR(20), cognome VARCHAR(30), data_nascita
DATE, comune_nascita VARCHAR(50)) RETURNS CHAR(16)
BEGIN
    -- algoritmo omissso ☺
    RETURN 'AAABBB11C22D333E';
END$$

-- se il codice fiscale inserito è nullo, lo calcoliamo automaticamente
CREATE TRIGGER fix_cf BEFORE INSERT ON impiegati FOR EACH ROW
BEGIN
    IF NEW.codice_fiscale IS NULL THEN
        -- non abbiamo dati sufficienti per il calcolo, il codice che segue è solo un esempio...
        SET NEW.codice_fiscale=calcola_codice_fiscale(NEW.nome,NEW.cognome,null,null);
    END IF;
END$$

-- se il codice fiscale è vuoto (dopo l'eventuale fix), generiamo un errore
-- solo a partire da MySQL 5.7.2 si possono inserire più trigger sullo stesso evento
CREATE TRIGGER check_cf_ins BEFORE INSERT ON impiegati FOR EACH ROW FOLLOWS fix_cf
BEGIN
    IF NEW.codice_fiscale = '' THEN
        SIGNAL SQLSTATE '45000' SET message_text = 'Il codice fiscale può essere nullo, ma non
vuoto!';
    END IF;
END$$

-- se il codice fiscale viene "svuotato" in aggiornamento, ripristiniamo il precedente, se
disponibile
CREATE TRIGGER check_cf_upd BEFORE UPDATE ON impiegati FOR EACH ROW
BEGIN
    IF (NEW.codice_fiscale = '' OR NEW.codice_fiscale IS NULL) AND
        (OLD.codice_fiscale IS NOT NULL AND OLD.codice_fiscale <> '') THEN
        SET NEW.codice_fiscale = OLD.codice_fiscale;
    END IF;
END$$
DELIMITER ;
```

SQL: Transazioni





Transazioni

- › In un database le operazioni **sono sempre raggruppate in *transazioni***.
- › Una transazione è un'unità di lavoro atomica che può essere applicata (*commit*) o cancellata (*rollback*) in blocco dal database.
- › In questo modo, è possibile evitare che il database resti in uno stato inconsistente a seguito, ad esempio, di un errore, annullando tutte le operazioni coinvolte in blocco.
- › Le transazioni, in DBMS avanzati come MySQL, hanno una serie di proprietà denominate **ACID**:
 - **Atomicity** (*atomicità*): la transazione è indivisibile e la sua esecuzione deve essere o completa o nulla,
 - **Consistency** (*consistenza*): quando la transazione termina il database deve essere in un altro stato consistente, cioè non deve violare alcun vincolo di integrità,
 - **Isolation** (*isolamento*): ogni transazione deve essere eseguita in modo isolato e indipendente dalle altre transazioni eventualmente attive sullo stesso database,
 - **Durability** (*persistenza*): quando una transazione termina con un *commit*, i cambiamenti da essa apportati devono essere definitivi.



Transazioni

- › Per avviare una transazione, si usa il comando **START TRANSACTION**
 - In MySQL, che *non supporta le transazioni nidificate*, eseguire una START TRANSACTION mentre una transazione è già attiva comporta un COMMIT esplicito: *la transazione precedente viene confermata e ne viene aperta una nuova*.
- › Per applicare in modo definitivo i cambiamenti effettuati all'interno di una transazione attiva e chiudere la transazione stessa si usa il comando **COMMIT**
- › Per annullare tutti i cambiamenti effettuati all'interno di una transazione attiva e chiudere la transazione stessa si usa il comando **ROLLBACK**
- › Molti DBMS dispongono di una caratteristica di default detta *autocommit*, che applica automaticamente le modifiche apportate da ogni statement che non sia esplicitamente racchiuso in una transazione.
 - In pratica, se non si apre una transazione, ogni statement verrà racchiuso in una transazione seguita da un COMMIT implicito.
- › In MySQL l'autocommit è per default attivo e si può disabilitare col comando **SET autocommit = 0**
 - Disabilitando l'autocommit, il DBMS **manterrà comunque automaticamente aperta una transazione** che però **dovrà essere chiusa esplicitamente** con i normali COMMIT e ROLLBACK, dopo i quali verrà automaticamente aperta una nuova transazione e così via.
 - Creare una transazione in modo esplicito con START TRANSACTION **disabilita l'autocommit in maniera temporanea**, finché la transazione attivata non viene chiusa.



Esempi

```
-- eseguendo questo blocco di istruzioni, il database non verrà modificato!
START TRANSACTION;
INSERT INTO impiegati(ID, codice_fiscale, nome, cognome, mansione, stipendio,
IDreparto, ufficio) VALUES ('240', null, 'Giuseppe', 'Verdi', 'impiegato', '8',
'0', '1');
ROLLBACK;

-- questa procedura esegue due inserimenti, ma se uno fallisce anche l'altro viene
annullato
DELIMITER $$
CREATE PROCEDURE entrambi_o_nessuno()
BEGIN
    DECLARE all_ok BOOLEAN;
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET all_ok = false;
    SET all_ok = true;
    START TRANSACTION;
    INSERT INTO impiegati (ID, codice_fiscale, nome, cognome, mansione, stipendio,
IDreparto, ufficio) VALUES (240, null, 'Pinco', 'Pallino', 'impiegato', 9, 0, 1);
    INSERT INTO impiegati (ID, codice_fiscale, nome, cognome, mansione, stipendio,
IDreparto, ufficio, IDsuperiore) VALUES (241, null, 'Paolino', 'Paperino',
'impiegato', 5, 1000, 1, 240);
    IF NOT all_ok THEN ROLLBACK;
    ELSE COMMIT;
    END IF;
END$$
DELIMITER ;
```

SQL Avanzato: Common Table Expressions

Realizzare query ricorsive in SQL





SQL Avanzato: strutture ricorsive

- › In molti casi è possibile che un database codifichi delle gerarchie, ad esempio con la tabella che segue

```
CREATE TABLE hier(  
id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
nome VARCHAR(50), parent INT DEFAULT NULL,  
CONSTRAINT autoref FOREIGN KEY (parent) REFERENCES hier(id)  
ON DELETE SET NULL ON UPDATE CASCADE)
```

- › Possiamo modellare una gerarchia tipo

root > sub1, sub2 > sub21 > sub211

inserendo i dati (con la sintassi INSERT ESTESA di MySQL):

```
INSERT INTO hier(id, parent, nome) VALUES (1,null,"root"), (2,1,"sub1"),  
(3,1,"sub2"), (4,3,"sub21"), (5,4,"sub211")
```




SQL Avanzato: query ricorsive?

- › Come è possibile attraversare questa gerarchia, per rispondere ad esempio alla domanda *"dato un ID, il nome dell'elemento corrispondente e di tutti gli elementi che ne discendono"*?
- › **Sappiamo che SQL non è ricorsivo, quindi in teoria non è possibile scrivere uno statement che richiama se stesso, come faremmo se dovessimo risolvere questa interrogazione con un linguaggio di programmazione.**
- › Tuttavia, abbiamo due possibili soluzioni
 - Scrivere una procedura ricorsiva, possibilmente usando tabelle temporanee di appoggio,
 - Utilizzare una *common table expression* ricorsiva (disponibile in MySQL solo dalla versione 8.0).



SQL Avanzato: procedure ricorsive

- › Si tratta di una soluzione articolata che si basa sull'uso di tabelle temporanee di appoggio
 - Le tabelle temporanee possono essere create in MySQL come *temporary tables* in memoria, per un accesso più rapido, tuttavia hanno delle caratteristiche particolari, e il loro uso richiede un continuo "riversamento di dati", come vedremo.
- › Il codice che segue risolve la query appena specificata.



Esempi

```
DROP PROCEDURE IF EXISTS hier_traverse;
DELIMITER $$
CREATE PROCEDURE hier_traverse(IN base_id INT)
BEGIN
    DECLARE v_done TINYINT UNSIGNED; DECLARE v_depth SMALLINT UNSIGNED;

    SET v_done= 0;SET v_depth= 0;

    DROP TEMPORARY TABLE IF EXISTS all_levels; DROP TEMPORARY TABLE IF EXISTS cur_level;

    -- tabella risultato con tutti i nodi via via estratti
    CREATE TEMPORARY TABLE all_levels ENGINE=MEMORY SELECT parent, id, 0 as depth FROM hier WHERE id = base_id;

    -- tabella di appoggio con tutti i nodi del solo livello corrente (v_depall_levels)
    CREATE TEMPORARY TABLE cur_level ENGINE=MEMORY SELECT parent,id FROM all_levels;

    -- questo loop costituisce una ricorsione "srotolata"
    WHILE NOT v_done DO
    BEGIN
        -- verifichiamo se esistono nodi discendenti dal livello corrente
        IF ((SELECT COUNT(*) FROM hier INNER JOIN cur_level ON hier.parent = cur_level.id)>0) THEN
        BEGIN
            -- inseriamo i successori del livello corrente nel risultato
            INSERT INTO all_levels SELECT hier.parent, hier.id, v_depth + 1
                FROM hier INNER JOIN cur_level ON hier.parent = cur_level.id;
            -- incrementiamo il livello
            SET v_depth = v_depth + 1;
            -- aggiorniamo la tabella con i nodi del solo (nuovo) livello corrente
            TRUNCATE TABLE cur_level;
            INSERT INTO cur_level SELECT parent,id FROM all_levels WHERE depth = v_depth;
        END;
        ELSE
        BEGIN
            SET v_done = 1;
        END;
        END IF;
    END WHILE;

    -- risultato finale
    SELECT hier.nome FROM all_levels INNER JOIN hier on all_levels.id = hier.id ORDER by all_levels.depth, all_levels.id;
    -- rimozione dati temporanei
    DROP TEMPORARY TABLE IF EXISTS all_levels; DROP TEMPORARY TABLE IF EXISTS cur_level;
END$$
DELIMITER ;

CALL hier_traverse(1); -- esempio d'uso
```



SQL Avanzato: le Common Table Expressions

- › Una CTE (*Common Table Expression*) è una tabella temporanea, ottenuta come risultato di una query ed utilizzata per eseguire un altro statement SQL

WITH *nome_cte* as (*query_cte*) *query*

- › In MySQL, le CTE sono disponibili solo dalla versione 8, e possono sostituire viste o subquery in vari contesti.

Le CTE possono però essere anche ricorsive, e in questo caso la query che le definisce può richiamare la CTE stessa (ovviamente bisogna fornire un adeguato caso base per la ricorsione)

WITH RECURSIVE *nome_cte* as (*query_cte*) *query*



Esempi

```
-- query con CTE banale
WITH th
AS (SELECT id, CONCAT(id,'-',nome) as nome FROM hier)
SELECT nome FROM th WHERE id=4
-- in questo caso era senz'altro più sensato scrivere
SELECT CONCAT(id,'-',nome) as nome FROM hier WHERE id=4;
```

```
-- qui invece risolviamo l'interrogazione proposta nelle slides precedenti
-- con MOLTO meno codice...
WITH RECURSIVE th AS (
-- caso base
SELECT id, parent, nome FROM hier WHERE id=1 -- parametro
UNION ALL
-- caso ricorsivo
SELECT hier.id,hier.parent,hier.nome
FROM hier INNER JOIN th ON (hier.parent=th.id))
SELECT nome FROM th
```

SQL e Linguaggi di Programmazione

Interfacciarsi con un DBMS ed eseguire comandi SQL da codice scritto in Java e PHP





Interfacciamento con i linguaggi di programmazione

- › Tipicamente una base di dati sarà utilizzata da una o più applicazioni client, che si interfaceranno ad essa per inserire, modificare ed estrarre informazioni (oltre che, in casi particolari, anche per modificare le struttura della base di dati stessa).
- › In generale, l'interazione tra DBMS e programma client utilizza SQL come "lingua franca", ma necessita comunque di particolari strutture, diverse per ciascun linguaggio/piattaforma software, che servono ad adattare i valori restituiti da SQL (tabelle, informazioni di stato, eccezioni, ...) alle strutture dati utilizzabili all'interno del codice.
- › Vedremo ora brevemente come ci si interfaccia con i DBMS in PHP e Java.
- › *In queste lezioni studieremo solo l'interfaccia base (a basso livello) verso i DBMS: i linguaggi a oggetti, come PHP e Java, dispongono anche di modalità di interfacciamento ad alto livello basate sull'ORM (object-relational mapping)*



Interfacciamento con i linguaggi di programmazione

I Cursori

- › Un concetto utile da comprendere, su cui si basano (anche se non esplicitamente) molte delle modalità di accesso ai DBMS è quello di cursore, che abbiamo già introdotto parlando della procedure SQL.
- › Poiché nella maggior parte dei linguaggi di programmazione non esiste il concetto di tabella dati e poiché, ad esempio, trasformare una tabella dati del DBMS in un array comporterebbe un enorme spreco di memoria nell'applicazione client, spesso i risultati di una query SQL vengono gestiti con l'aiuto di un cursore.
- › Questo oggetto non è altro che un puntatore che può muoversi all'interno di un insieme di record (il risultato dell'interrogazione), permettendo di leggere i record stessi uno alla volta.
- › In pratica, un cursore viene creato associandovi una particolare query.
 - Esistono istruzioni che permettono di muovere il cursore avanti e indietro sull'insieme di record risultanti.
 - E' possibile usare il cursore per leggere i valori di ciascuna colonna del record corrente, cioè quella correntemente puntata dal cursore.



JAVA e DBMS

- › L'accesso ai dati in Java si effettua tramite il JDBC (Java DataBase Connectivity), le cui classi sono contenute nel **package java.sql**
- › E' necessario che il **driver JDBC** per il DBMS in uso sia disponibile nel classpath di Java.
 - Per MySQL, il driver si chiama *connector/j* ed è scaricabile all'indirizzo <https://dev.mysql.com/downloads/connector/j>
 - Se usate Maven, potete individuare la dipendenza da aggiungere al vostro pom cercando l'ultima versione disponibile del driver nel repository <https://mvnrepository.com/artifact/mysql/mysql-connector-java>
- › L'accesso a un database con il JDBC si articola in sette fasi:
 1. Caricamento driver
 2. Connessione
 3. Creazione statement (query)
 4. Esecuzione query
 5. Lettura risultati (*se la query li prevede*)
 6. Chiusura risultati e statement
 7. Chiusura connessione
- › Poiché mantenere aperti risultati, statement e soprattutto connessioni **consuma risorse**, di solito le istruzioni JDBC vengono poste in un blocco *try* e i punti 6 e 7 vengono eseguiti all'interno del relativo blocco *finally*.
- › Tutte le istruzioni JDBC, in caso di errore, sollevano eccezioni derivate da `SQLException`.



JAVA e DBMS: JDBC

Connessione

- › Nelle versioni del Connector/J **precedenti alla 8** il driver doveva essere caricato esplicitamente facendo riferimento alla classe che lo implementa **`Class.forName("com.mysql.cj.jdbc.Driver");`**
Questo non è più necessario (anche se non produce errori) nelle versioni correnti, che vengono caricate da Java automaticamente.
- › Si procede alla creazione dell'oggetto **Connection** tramite il metodo statico `getConnection` della classe *DriverManager*:
`Connection con = DriverManager.getConnection(connectionString, username, password);`
 - I parametri *username* e *password* identificano l'utente con cui ci si sta connettendo al DBMS (e che dovrà ovviamente avere i permessi necessari alle operazioni che intendete eseguire).
 - La *connectionString* è la stringa di connessione JDBC. Questa stringa, la cui forma generale è **`jdbc:dbms://database?parametri`** ha un contenuto che varia a seconda del DBMS in uso.
 - › Nel caso di MySQL, contiene l'indirizzo al quale risponde il server MySQL (*host*) e il nome del *database* sul quale si desidera operare.
 - › La *query string* (formata da coppie chiave=valore separate da '&') *parametri* contiene parametri ulteriori da usare per la connessione.
 - › Con MySQL, a partire dal Connector/J versione 8, se sul server non è impostata una *timezone*, dovrete specificarla come parametro scrivendo, ad esempio, **`serverTimezone=Europe/Rome`**. Inoltre, se dovete eseguire delle chiamate a procedura e non avete permessi elevati, è utile specificare il parametro **`noAccessToProcedureBodies=true`**.
 - › Una stringa tipo per la connessione a MySQL per un database locale chiamato «azienda» potrebbe quindi essere
`jdbc:mysql://localhost:3306/azienda?noAccessToProcedureBodies=true&serverTimezone=Europe/Rome`
- › Terminato l'uso della base di dati, la connessione **deve essere sempre chiusa** chiamando il metodo *close* della *Connection*:
`con.close();`



JAVA e DBMS: JDBC

Creazione statement ed esecuzione query

- › Si crea un oggetto **Statement** sulla connessione, usando il metodo *createStatement*
Statement stmt = con.createStatement();
- › Per le query di selezione, si invia la query SQL, sotto forma di stringa, al DBMS tramite lo *Statement* creato e il suo metodo *executeQuery*
ResultSet rs = stmt.executeQuery("query");
Per le query di modifica e aggiornamento (DDL e DML), si usa invece il metodo *executeUpdate*
int affected = stmt.executeUpdate("query");
 - L'oggetto restituito da *executeQuery*, di tipo **ResultSet**, permette di navigare tra i risultati della query.
 - Il valore restituito da *executeUpdate* è un intero che rappresenta il numero di record interessati (inseriti, aggiornati, ecc.) dalla query stessa
- › In ogni caso, una volta eseguita la query e prelevati gli eventuali risultati, si libera lo spazio a loro riservato chiamando il metodo *close* dello *Statement* ed eventualmente del *ResultSet*:
rs.close(); stmt.close();



JAVA e DBMS: JDBC

Lettura dei risultati

- › Per le query che ritornano tabelle di risultati, tramite il **ResultSet** restituito dal metodo *executeQuery* è possibile leggere le colonne di ciascun record restituito da una query di selezione.
 - I record devono essere letti uno alla volta: in ogni momento, il *ResultSet* punta (tramite un cursore) a uno dei record restituiti (record corrente).
- › Per spostare il cursore del *ResultSet* al record successivo, si usa il metodo *next*. Il metodo restituisce false quando i record sono esauriti e viene quindi solitamente utilizzato in un loop while:
while (rs.next()) { ... }
- › I valori dei vari campi del record corrente possono essere letti tramite i metodi *getX(nome_colonna)*, dove *X* è il tipo base Java per il dato da estrarre (ad esempio *getString*, *getInt*, ...) e *nome_colonna* è il nome del campo del record da leggere.
rs.getString("nome_colonna")
 - in alternativa è possibile usare i metodi *getX(indice_colonna)* per accedere alle colonne in base al loro ordine (numero con base 1) nell'output.



JAVA e DBMS: JDBC

Prepared Statements

- › Tuttavia, quella appena vista non è la modalità consigliata per eseguire una query. Per motivi di efficienza e sicurezza (che discuteremo dopo), è sempre necessario usare i **prepared statements**.
- › Un prepared statement è uno statement SQL che viene fatto **compilare dal DBMS** ed eseguito successivamente.
 - Un prepared statement può contenere **parametri**, che verranno rimpiazzati con i loro valori effettivi solo al momento dell'esecuzione, permettendo la scrittura di uno statement generico e il suo riuso con parametri diversi.

- › Invece di costruire uno statement ed eseguirlo, magari inserendoci un parametro tramite la semplice concatenazione di stringa:

```
Statement stmt = con.createStatement();
```

```
ResultSet rs = stmt.executeQuery("SELECT * FROM t WHERE ID="+id);
```

1. Prepariamo lo statement passando la query al metodo *prepareStatement* della *Connection*
PreparedStatement stmt = con.prepareStatement("SELECT * FROM t WHERE ID=?")
 - La query non viene eseguita, solo *compilata*
 - I *parametri* sono indicati nella query con dei punti interrogativi. *Da notare che per le costanti stringa non è necessario mettere il punto interrogativo tra virgolette.*
2. Assegniamo dei valori ai parametri usando i metodi *setX(indice_parametro, valore)* del *PreparedStatement*;
stmt.setInt(1, id)
 - L'*indice_parametro* punta al punto interrogativo da sostituire (1=primo punto interrogativo)
 - Il tipo X della set corrisponde al tipo del *valore*.
3. Eseguiamo lo statement risultante chiamando uno dei metodi già visti per lo *Statement* semplice, che in questo caso non accettano alcun parametro:
ResultSet rs = stmt.executeQuery();



JAVA e DBMS: JDBC

Chiamata a procedure

- › Si compila un oggetto **CallableStatement** sulla connessione, usando il metodo *prepareCall*:
CallableStatement stmt = con.prepareCall("{call procedura(?,?)}");
 - Notare la forma particolare della call in questo caso, racchiusa tra graffe
- › Come al solito, si impostano i valori dei parametri della procedura **di tipo IN o INOUT** (se presenti) sullo statement con i metodi *setX(indice_parametro, valore)*:
stmt.setInt(1, v)
- › Se la procedura prevede dei **parametri OUT o INOUT**, bisogna dichiararli **stmt.registerOutParameter(2, Types.VARCHAR);**
 - Da notare che va specificato il tipo DBMS dei valori di output attesi
- › Si esegue la chiamata tramite il metodo *execute*
stmt.execute()
- › Se la procedura genera una tabella dati, è possibile accedervi usando il metodo *getResultSet*:
ResultSet rs = stmt.getResultSet();
- › Se ci sono parametri di tipo OUT o INOUT, possiamo leggerne i valori calcolati usando i metodi *getX(indice_parametro)*:
s.getString(2)



PHP e DBMS: MySQLi

- › Il linguaggio di scripting PHP è molto diffuso per la programmazione server-side di siti internet. Per questo motivo l'interfacciamento con i DBMS è una sua funzionalità nativa.
- › In PHP esistono set di istruzioni differenti per interfacciarsi con tutti i più noti DBMS. Descriveremo qui di seguito una tipica interazione con un DBMS MySQL, limitandoci alle istruzioni necessarie a eseguire delle query. Per gli altri DBMS le istruzioni e le procedure sono molto simili.
- › In particolare, vediamo l'estensione **MySQLi**, che permette di manipolare il database tramite oggetti di classe *mysqli*.
- › L'accesso a un database in PHP si articola in quattro fasi:
 1. Connessione
 2. Esecuzione query
 3. Lettura risultati (se la query li prevede)
 4. Chiusura connessione
- › In caso di errore, è possibile leggere il relativo codice e il messaggio testuale accedendo ai seguenti campi degli oggetti di classe *mysqli*:
 - **\$mysqliobj->connect_errno** e **\$mysqliobj->connect_error**
per gli errori di connessione (metodo connect)
 - **\$mysqliobj->errno** e **\$mysqliobj->error**
per gli errori dell'ultima istruzione eseguita sul DBMS.



PHP e DBMS: MySQLi

1. Per prima cosa, è necessario connettersi al DBMS e selezionare il database su cui operare. A questo scopo, si crea un oggetto **mysqli**:
`$mysqliobj = new mysqli("host", "user", "password", "database")`
 - *host* è l'indirizzo al quale risponde il server MySQL, ad esempio localhost:3386
 - *database* è il nome del database sul quale si desidera operare
 - *username* e *password* sono le credenziali di un utente registrato nel DBMS che ha accesso al *database* indicato
2. La query SQL viene passata al DBMS come una semplice stringa di testo usando il metodo **query**:
`$res = $mysqliobj->query("query")`
 - dove *query* è un'istruzione SQL
 - Il valore di ritorno (\$res) è FALSE in caso di errore.
 - Per le istruzioni SELECT, la chiamata al metodo query ritorna un oggetto che permette di analizzare i risultati ottenuti
 - Per tutte le altre istruzioni SQL, in caso di esecuzione con successo, la chiamata ritorna TRUE
3. E' possibile iterare sulle righe restituite dalla query usando il metodo **fetch_assoc**:
`while ($row = $res->fetch_assoc()) {...}`
 - La variabile \$row è un array associativo indicizzato dai nomi delle colonne del record corrente.
4. Terminato l'uso della base di dati, si può chiudere la connessione corrispondente chiamando il metodo **close**:
`$mysqliobj->close()`



PHP e DBMS: MySQLi

Prepared Statements

› Invece di costruire lo statement ed eseguirlo:

```
$res = $mysqliobj->query("SELECT * FROM t WHERE ID=$id")
```

1. Lo prepariamo

```
$stmt = $mysqliobj->prepare("SELECT * FROM t WHERE ID=?")
```

(dove ogni "?" costituisce un parametro)

2. Dopodichè assegniamo i parametri

```
$id = 2; $stmt->bind_param("i", $id);
```

(dove "i" indica che stiamo sostituendo un parametro di tipo intero col valore della variabile \$id)

3. E infine eseguiamo lo statement risultante

```
$stmt->execute(); $res = $stmt->get_result();
```

ora \$res contiene i risultati della query, come nel caso non-prepared.

› Nota: se stiamo sostituendo un parametro di tipo stringa, la query non dovrà contenere alcun delimitatore: ci penserà il DBMS! Ad esempio

```
$stmt = $mysqliobj->prepare("SELECT * FROM u WHERE nome=?")
```

```
$nome = "pippo"; $stmt->bind_param("s", $nome);
```



PHP e DBMS: PDO

- › Per poter interagire con i DBMS in maniera più astratta, similmente a quanto avviene in Java col JDBC, PHP mette a disposizione anche un altro sistema di accesso, l'estensione **PDO**.
- › Come nel caso di JDBC, anche per PDO è necessario che PHP abbia disponibili i driver per il DBMS utilizzato, ad esempio l'estensione **PDO_MySQL**.
- › Per il resto, l'interazione è simile, anche se gli oggetti e il codice sono differenti.
- › In caso di errore, per default PDO genera eccezioni di tipo **PDOException**, che possono essere catturate e gestite normalmente.



PHP e DBMS: PDO

1. Per prima cosa, è necessario connettersi al DBMS e selezionare il database su cui operare. A questo scopo, si crea un oggetto **PDO** passandogli una *connection string* (simile a quella JDBC) e le credenziali di connessione:
`$conn = new PDO("mysql:host=host;dbname=database", "user", "password");`
 - *host* è l'indirizzo al quale risponde il server (MySQL in questo esempio), ad esempio localhost:3386
 - *database* è il nome del database sul quale si desidera operare
 - *username* e *password* sono le credenziali di un utente registrato nel DBMS che ha accesso al *database* indicato
2. La query SQL viene passata al DBMS come una semplice stringa di testo usando il metodo **query**:
`$stmt = $conn->query("query")`
 - dove *query* è un'istruzione SQL
 - Il valore di ritorno (`$stmt`) è FALSE in caso di errore.
 - Per le istruzioni SELECT, la chiamata al metodo query ritorna un oggetto che permette di analizzare i risultati ottenuti
3. E' possibile iterare sulle righe restituite dalla query usando il metodo **fetch**:
`while ($row = $stmt->fetch(PDO::FETCH_ASSOC)) {...}`
 - La variabile `$row` è un array associativo indicizzato dai nomi delle colonne del record corrente.
4. Terminato l'uso della base di dati, si può chiudere la connessione corrispondente semplicemente eliminando ogni riferimento ad essa:
`$conn = null`



PHP e DBMS: PDO

Prepared Statements

› Invece di costruire lo statement ed eseguirlo:

```
$stmt = $conn->query("SELECT * FROM t WHERE ID=$id")
```

1. Lo prepariamo

```
$stmt = $conn->prepare("SELECT * FROM t WHERE ID=:id")  
(dove ogni "?" costituisce un parametro)
```

2. Dopodichè assegniamo i parametri

```
$id = 2; $stmt->bindParam(":id", $id);
```

3. E infine eseguiamo lo statement risultante

```
$stmt->execute();
```

ora `$stmt` permette di accedere ai risultati della query, come nel caso non-prepared.

Sicurezza nei DBMS

*Potenziati problemi di sicurezza derivanti
dall'interfacciamento tra DBMS e
linguaggi di programmazione*





Alcune considerazioni sulla sicurezza

- › I DBMS sono spesso usati nelle **applicazioni web**, cioè in quelle applicazioni installate su un web server e utilizzate tramite un web browser.
- › Nella applicazioni web, **una delle più comuni problematiche di sicurezza è legata proprio all'interfacciamento scorretto con la base di dati**, ed è nota come **SQL Injection**.
- › Potete vedere, ad esempio, come si piazza la SQL Injection nella top ten degli attacchi alle applicazioni web: <https://owasp.org/www-project-top-ten>
- › Cerchiamo di capirla e di evitarla!
- › Per prima cosa, consideriamo che l'injection può avvenire solo quando **una query SQL viene "costruita", sotto forma di stringa, sulla base di parametri provenienti dall'utente**, magari inviati tramite un modulo su una pagina HTML.



SQL Injection

Schema di attacco

- › Ecco un esempio di SQL Injection in PHP. Immaginiamo di avere un modulo nel quale l'utente inserisce l'identificativo di documento, ricevendone in risposta il titolo.
- 1. Lo script PHP riceverà il **numero inviato dall'utente** (non preoccupatevi di COME a questo livello):
\$id = \$_GET('id')
- 2. Quindi **costruirà la query di estrazione con una semplice concatenazione di stringhe**:
\$query = "SELECT titolo FROM documenti WHERE ID=\$id"
- 3. Fin qui tutto bene. Se l'utente immette, ad esempio, "2" come identificativo, avremo la \$query (corretta)
SELECT titolo FROM documenti WHERE ID=2
- › Ma se un utente malizioso inviasse, come identificativo, la stringa **"2 OR 1=1"**, la \$query risultante sarebbe
SELECT titolo FROM documenti WHERE ID=2 OR 1=1
...e in questo caso estrarremmo **i titoli di tutti i documenti!**
- › Si può fare di peggio: se l'utente inviasse come identificativo la stringa **"2; DROP TABLE documenti"** la query risultante
SELECT titolo FROM documenti WHERE ID= 2; DROP TABLE documenti
verrebbe interpretata come la concatenazione di due query, la seconda delle quali **cancella l'intera tabella!**
- › Per finire, un esempio classico per un modulo di login nel quale l'utente inserisce username e password. La query che verifica se l'utente è presente nel database potrebbe essere (notate l'uso delle virgolette):
\$query = "SELECT * FROM utenti WHERE username = '\$username' AND password = '\$password'"
- › Con un input normale, avremmo query del tipo
SELECT * FROM utenti WHERE username = 'pinco' AND password = 'jd89389erh'
- › Se invece inserissimo come password la stringa **" OR ""** otterremmo la query
SELECT * FROM utenti WHERE username = 'pinco' AND password = "" OR ""
che estrae le informazioni di tutti gli utenti!
- › Altri esempi? Guardate su https://owasp.org/www-community/attacks/SQL_Injection



SQL Injection

Difesa

- › Come difendersi?
- › **Sanitizzando e controllando** i parametri che vengono dall'esterno: ad esempio,
 - se mi aspetto un **numero** (come nei primi esempi della slide precedente), devo verificare che sia un numero prima di immetterlo nella query.
 - se mi aspetto una **stringa**, devo rimuovere o sottoporre a *escaping* i caratteri pericolosi, come i delimitatori.
- › Possiamo fare in modo che sia (anche) il DBMS ad eseguire questi controlli, tramite i ***prepared statements***.
- › Infatti, prima di eseguire il prepared statement, **i parametri vengono rimpiazzati da valori**, ma non agendo sulla stringa SQL, bensì **passandoli al DBMS stesso e specificandone il tipo**.
- › In base al tipo dichiarato, **il DBMS eseguirà i controlli e le correzioni necessarie**.