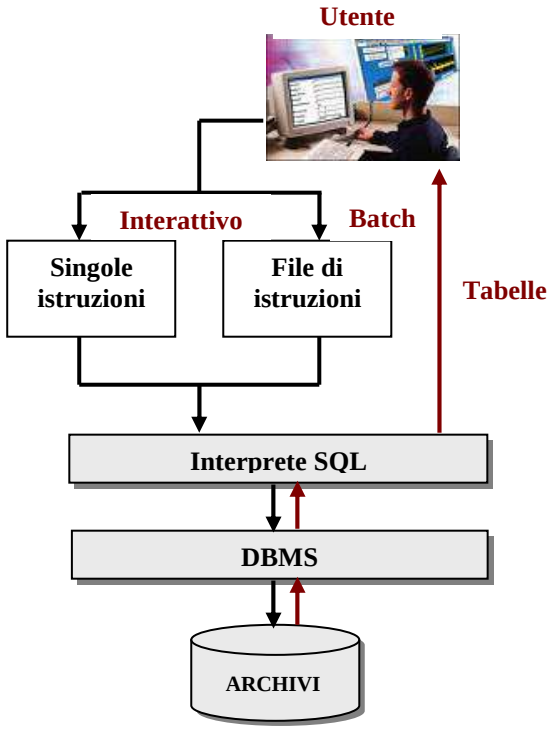
16. IL LINGUAGGIO SQL

Il **linguaggio SQL** (***S**tructured **Q**uery **L**anguage*) è un linguaggio **non procedurale** (in quanto non richiede la descrizione dei passi elementari di elaborazione) **o di tipo dichiarativo** (in quanto è un tipo di linguaggio in cui le istruzioni si limitano a descrivere "cosa" si vuol fare e non "come" farlo). E' ormai da tempo uno degli standard tra i linguaggi per basi di dati relazionali.

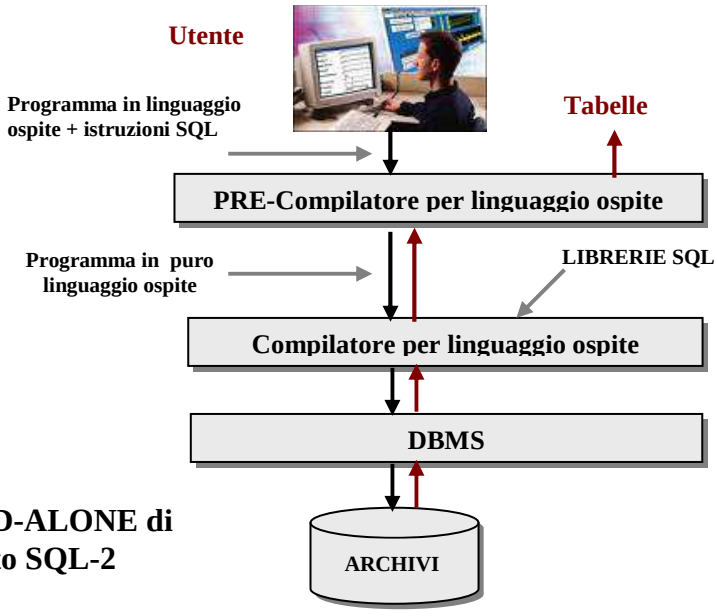
Il linguaggio **SQL** assolve alle funzioni di:
- **DDL** (***D**ata **D**efinition **L**anguage*) che prevede istruzioni per definire la struttura delle relazioni della base di dati. Serve quindi a creare *tabelle, vincoli*, *viste* ed *indici*;
- **DML** (***D**ata **M**anipulation **L**anguage*) che prevede istruzioni per manipolare i dati contenuti nelle diverse tabelle. Serve in particolare per *inserire*, *cancellare* e *modificare* ennuple;
- **DCL** (***D**ata **C**ontrol **L**anguage*) che prevede istruzioni per controllare il modo in cui le operazioni vengono eseguite. Consente di creare e cancellare gli utenti, gestire il *controllo degli accesi* a più utenti ed i *permessi* agli utenti autorizzati.

Il linguaggio **SQL** può essere usato in:

- **<u>modalità stand-alone</u>**: in questa modalità può essere classificato come **query language interattivo.** I comandi vengono inviati al sistema operativo in modo **interattivo** (utilizzando un'apposita interfaccia grafica con menù, finestre ed icone) **oppure batch** (creando file di istruzioni da eseguire in gruppo).
In entrambe le modalità viene invocato **l'interprete** SQL.
Le interrogazioni o query vengono composte utilizzando le operazioni dell'algebra relazionale ottenendo come risultato sempre una tabella che rappresenta una relazione.



- **<u>modalità embedded</u>**: in questa modalità è possibile utilizzare comandi SQL all'interno di istruzioni di altri linguaggi (esempio C, Java, C++) detti "linguaggi ospite".
In linea di massima un programma scritto in *linguaggio ospite compilato* (che incorpora comandi SQL al suo interno) subirà un primo processo di *precompilazione* (che si occupa di tradurre le sole istruzioni SQL in istruzioni in linguaggio ospite compilato) seguito da una vera e propria *compilazione* (che si occupa di tradurre le istruzioni specifiche del linguaggio ospite).



Noi faremo riferimento alla versione STAND-ALONE di SQL nel suo standard adottato nel 1992 detto SQL-2

IDENTIFICATORI E TIPI DI DATI

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

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

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

Gli **identificatori** utilizzati per i nomi di tabelle e di attributi devono:
-avere lunghezza max di 18 caratteri;
-iniziare con una lettera;
-contenere come unico carattere speciale l'underscore ossia ' _ ';

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

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

<NomeTabella>.<NomeAttributo>

I **tipi di dato** utilizzabili per gli attributi sono riassunti nella tabella a pag. 91 del libro di testo anche se soccorre precisare che in alcune versioni tali tipi potrebbero essere differenti.

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

Nelle **espressioni** possono anche essere usati i seguenti operatori
- *aritmetici* (+, -, /, *);
- *relazionali* (<, >, <=, >=, < >, =);
- *logici* (AND, OR, NOT)

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

ISTRUZIONE DEL DDL di SQL

Creare un nuovo database

Per creare un nuovo database useremo l'istruzione

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

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

Per eliminare un database

DROP DATABASE <NomeDB> ;

Selezionare un database

Una volta creato un database esso viene aggiunto dal DBMS agli altri database precedentemente creati.
Per impartire comandi SQL per uno specifico database occorre prima selezionarlo utilizzando il comando:

```
USE <NomeDB> ;
```

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

Creazione di una tabella e dei vincoli di integrità

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

```
CREATE TABLE <NomeTabella>
        (
        <NomeAttributo1>  <Tipo1>  >],
        <NomeAttributo2>  <Tipo2> [<VincoloAttributo2>],
        ……
        <NomeAttributoN>  <TipoN> [<VincoloAttributoN>],
        [<VincoloTabella>]
        ) ;
```

Nella definizione di una tabella sono presenti vincoli:

- per un **singolo attributo**;
- per un **gruppo di attributi**, detti anche **"vincoli di ennupla"**;
- per l'**integrità referenziale**.

<VincoloAttributo1>, <VincoloAttributo2>,…,<VincoloAttributoN> sono vincoli per un singolo attributo mentre <Vincolo Tabella> può essere un vincolo di ennupla o di integrità referenziale.

Vincoli per un singolo attributo:
Sono detti anche "vincoli di dominio" impostano limitazioni da specificare sul valore di un singolo attributo.
Sono impostati dalle clausole

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

Vincoli per un gruppo di attributi:
Impostano limitazioni da specificare sui valori di più attributi.
Possono essere impostati dalle seguenti clausole:

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

Vincoli di integrità referenziale e politiche di violazione:
Possono essere dichiarati con la seguente clausola:

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

Le colonne <Attributo1>,..,<AttributoN> rappresentano la chiave esterna e corrispondono alle colonne <Attr1>,..,<AttrN> che formano la chiave primaria della tabella <Nome Tabella>.
La parte opzionale successiva è relativa la tipo di politica da seguire in caso di violazione del vincolo referenziale durante una operazione di **cancellazione** o **modifica**.

Al momento della **cancellazione** di un attributo interessato da un vincolo referenziale (si specifica con la clausola ON DELETE) si può avere il seguente comportamento:

- con l'opzione **CASCADE**, vengono cancellate le righe corrispondenti;
- con l'opzione **SET NULL**, vengono impostate a NULL le righe corrispondenti;
- con l'opzione **SET DEFAULT**, vengono impostate al valore di default le righe corrispondenti;
- con l'opzione **NO ACTION**, non viene eseguita alcuna azione (impostazione di default se non specificata la clausola ON DELETE).

Al momento della **modifica** di un attributo interessato da un vincolo referenziale (si specifica con la clausola ON UPDATE) si può avere il seguente comportamento:

- con l'opzione **CASCADE**, vengono aggiornate le righe corrispondenti con il nuovo valore;
- con l'opzione **SET NULL**, vengono impostate a NULL le righe corrispondenti;
- con l'opzione **SET DEFAULT**, vengono impostate al valore di default le righe corrispondenti;
- con l'opzione **NO ACTION**, non viene eseguita alcuna azione (impostazione di default se non specificata la clausola ON DELETE).

Vedi esempio pag. 97

Creazione di un dominio

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

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

dove

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

Esistono anche istruzioni per:
- eliminare un dominio (**DROP DOMAIN**);
- modificare un dominio (**ALTER DOMAIN**);

Modificare una tabella

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

ALTER TABLE <NomeTabella>
ADD COLUMN <NomeColonna1> <Tipo1> | <NomeDominio>,
[BEFORE <NomeColonna2>**] ;**

oppure

ALTER TABLE <NomeTabella>
DROP COLUMN <NomeColonna> **;**

oppure

ALTER TABLE <NomeTabella>
MODIFY COLUMN(<NomeColonna> <NuovoTipoColonna> **) ;**

(N.B. In MS ACCESS al posto di MODIFY COLUMN occorre ALTER COLUMN)

dove
- l'istruzione **ADD COLUMN** permette di aggiungere alla tabella <NomeTabella> la nuova colonna <NomeColonna1> di <Tipo1> oppure <NomeDominio> eventualmente specificando la posizione dove inserirla (con l'istruzione opzionale **BEFORE**) rispetto ad un'altra colonna <NomeColonna2>;
- l'istruzione **DROP COLUMN** viene usata per eliminare la colonna <NomeColonna>della tabella <NomeTabella>;
- l'istruzione **MODIFY COLUMN** viene usata per modificare solo il tipo di una colonna non il suo nome.

Eliminare una tabella

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

DROP TABLE <NomeTabella> **[RESTRICT | CASCADE | SET NULL] ;**

La cancellazione di una tabella può provocare inconsistenze dovute al fatto che possono esserci tabelle collegate tramite chiavi esterne o più in generale tramite vincoli di integrità.
Per far fronte a tali evenienze si utilizzano:
- l'istruzione **RESTRICT** che non permette la cancellazione se la tabella da cancellare è legata tramite vincoli di integrità ad altre tabelle (comportamento da non specificare perché di default);
- l'istruzione **CASCADE** che da luogo ad una cancellazione ricorsiva in cascata di tutte le tabelle collegate;
- l'istruzione **SET NULL** che pone a NULL tutti i valori delle chiavi interessate.

Creazione di un indice

In SQL è possibile legare agli attributi di una tabella alcune **tabelle speciali dette indici.** Tali indici sono file contenenti le chiavi delle tabelle alle quali sono associati che permettono all'SQL di accelerare il processo di ricerca dei dati all'interno della tabella ove sono immagazzinati.
Gli indici sono estranei al modello relazionale dei dati e sono molto vicini al **modello fisico dei dati.**
L'istruzione per creare un indice per una tabella è:

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

dove
- se specificata la clausola opzionale **UNIQUE** crea un indice su attributi chiave;
- se NON specificata l'istruzione crea un indice su attributi NON chiave;

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

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

DROP INDEX <NomeIndice **>;**

Riassumiamo quanto visto finora per creare e gestire i vincoli in SQL.

Per esprimere i **vincoli interni o intrarelazionali** utilizziamo durante la creazione di una tabella:
- la clausola **NOT NULL** per esprimere vincoli sul valore di un attributo;
- la clausola **PRIMARY KEY** per esprimere vincoli sulle chiavi primarie;
- la clausola **UNIQUE** per esprimere vincoli sulle chiavi candidate;
- la clausola **CHECK** per esprimere vincoli di ennupla che coinvolgono uno o più attributi;

Per esprimere i **vincoli interrelazionali di integrità referenziali** utilizziamo durante la creazione di una tabella:

- la clausola **FOREIGN KEY … REFERENCES** per indicare le chiavi esterne;
- la clausola **NO ACTION, CASCADE, SET DEFAULT e SET NULL** per implementare una delle tre politiche:
    - *rifiuto* delle modifiche che violano un vincolo
    - *propagazione* a cascata delle modifiche
    - *impostazione al valore di default* delle chiavi delle tabelle interessate
    - *impostazione a NULL* delle chiavi delle tabelle interessate

Per esprimere i **vincoli interrelazionali generici** (quelli non referenziali) non si utilizzano specificazioni di elementi dello schema inserendole all'interno di altre istruzioni SQL.
Essi vengono tradotti utilizzando apposite istruzioni SQL le **asserzioni** che sono esse stesse elementi dello schema.
Per creare un'asserzione si utilizza l'istruzione **CREATE ASSERTION** che ha la seguente sintassi:

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

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

ISTRUZIONE DEL DML di SQL

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

Inserire una riga in una tabella

In SQL è possibile inserire i valori in una tabella utilizzando l'istruzione **INSERT INTO** la cui sintassi è:

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

Se non presente la lista degli attributi (essendo opzionale come indica la parentesi quadra) si intende che i valori specificati devono corrispondere in ordine. Tipo e numero a quelli specificati nella dichiarazione di <NomeTabella>.
Se presente la lista degli attributi di <NomeTabella> l'ordine. il tipo e il numero dovrà rispettare questa lista e per gli attributi omessi dalla lista si assume abbiano valore NULL (quindi non devono essere stati dichiarati NOT NULL in fase di creazione della tabella)

Modificare i valori delle righe in una tabella

In SQL è possibile aggiornare una o più righe di una tabella utilizzando l'istruzione **UPDATE** la cui sintassi è:

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

dove gli attributi di <NomeTabella> (solo quelli specificati nella clausola **SET**) vengono aggiornati con i valori delle corrispondenti espressioni in tutte le righe che soddisfano la clausola **WHERE** se è presente oppure tutte le righe della tabella se non è presente.

Cancellare le righe di una tabella

In SQL è possibile cancellare una o più righe di una tabella utilizzando l'istruzione **DELETE FROM** la cui sintassi è:

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

In questo modo si eliminano tutte le righe di <NomeTabella> che soddisfano la <Condizione> specificata nella clausola **WHERE** se essa presente, altrimenti tutte le righe di <NomeTabella> indistintamente se tale clausola dovesse mancare.

Ci sono DBMS, non è il caso di MYSQL, in cui può essere specificato l'asterisco (*) tra il comando DELETE e la clausola FROM.

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

Per reperire i dati il linguaggio SQL utilizza l'istruzione **SELECT** la cui potenza ed espressività è forse alla base del suo successo.
Il risultato di una qualunque interrogazione effettuata tramite la **SELECT** è a sua volta una *tabella* che viene mostrata a video oppure stampata, ma che può anche essere assegnata ad una variabile strutturata.

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

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

L'istruzione **SELECT** restituisce una **tabella** formata dagli attributi
<NomeAttributo1> [, <NomeAttributo2> , <NomeAttributoN>]
della tabella (o del prodotto cartesiano delle tabelle)
<NomeTabella1> [, <NomeTabella2>, … , <NomeTabellaK>]
ristretto alle righe che soddisfano l'eventuale **<Condizione>** espressa dalla clausola **WHERE** se presente, altrimenti se assente la <Condizione> si assume sempre vera.

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

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

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

Intestare le colonne della tabella risultato
Per *default* la tabella risultato di una **SELECT** ha come intestazione delle colonne il nome degli attributi della tabella.
Se si vuole dare un nome diverso ad ogni colonna del risultato si deve utilizzare la clausola **AS** chiamata **alias.**

Eseguire calcoli sulla tabella prodotta da una SELECT senza modificare il contenuto delle tabelle
E' possibile far eseguire all'istruzione **SELECT** il calcolo di una espressione sugli attributi mostrandola a video in una nuova colonna intestata con la clausola **AS**.
Il calcolo viene eseguito esternamente alla tabella senza modificare i dati in essa contenuti.

Esempio: Se vogliamo visualizzare una variazione del 10% degli stipendi dei dipendenti potremo scrivere
***SELECT** Cognome, Nome, StipendioNetto * 1,10 **AS** NuovoStipendio **FROM** Dipendente;*

Le operazioni relazionali in SQL

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

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

Esempio: Se vogliamo l'elenco di tutti i dipendenti che hanno StipendioNetto maggiore o uguale a 1000,00 si opera una selezione sulla tabella dipendente
***SELECT * FROM** Dipendente **WHERE** StipendioNetto ≥1000,00 ;*
che equivale secondo lo pseudocodice dell'algebra relazionale all'operazione

$\sigma_{StipendioNetto \geq 1000,00}$ ***(****Dipendente****)***

(B) L'operazione di **proiezione** (Π), che consente di ottenere una relazione contenente solo alcuni attributi della relazione di partenza, viene realizzata nel linguaggio SQL utilizzando l'istruzione **SELECT** specificando l'elenco degli attributi richiesti ma senza impostare la clausola WHERE.

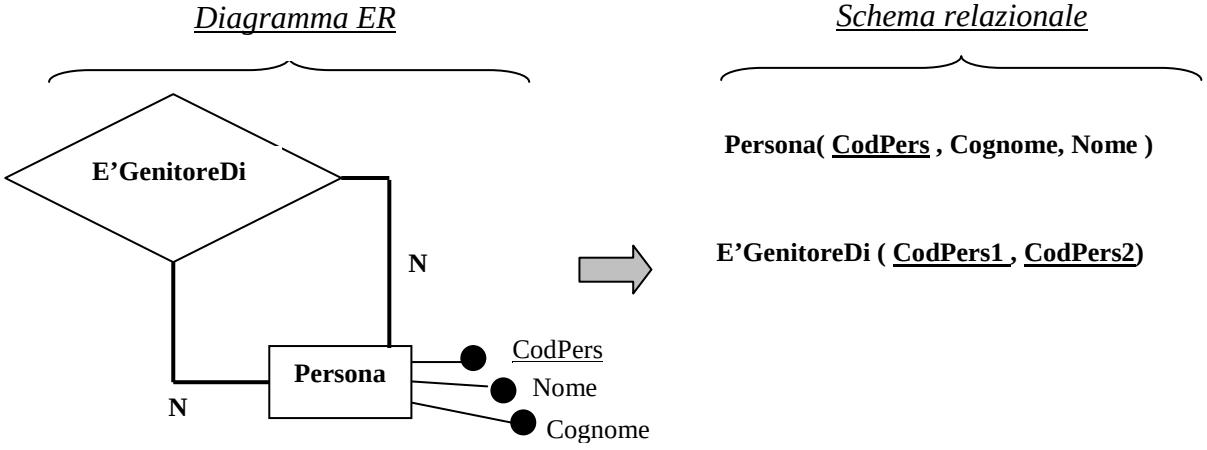
Esempio: Se vogliamo l'elenco di tutti i dipendenti visualizzando soltanto Cognome, Nome e StipendioNetto si opera una proiezione sulla tabella dipendente
***SELECT** Cognome, Nome, StipendioNetto **FROM** Dipendente;*
che equivale secondo lo pseudocodice dell'algebra relazionale all'operazione

$\Pi_{Cognome, Nome, StipendioNetto}$ ***(****Dipendente****)***

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

(D) Si realizza in pratica l'operazione di **giunzione naturale** o **join** (chiamata anche **inner-join** o **equi-join**) utilizzando il comando SELECT per effettuare il prodotto cartesiano come al punto (C) facendo cura di scegliere tutti gli attributi tranne quelli congiunti scritti una volta sola ed impostando grazie alla clausola **WHERE** l'uguaglianza dei valori tra i nomi degli attributi che corrispondono nelle due tabelle. Utilizzando gli **alias** (ossia la clausola **AS**) è possibile effettuare operazioni si **self-join** (ossia join sulla stessa tabella)

Esempio: Supponiamo di avere il seguente diagramma ER che dettaglia l'associazione èGenitoreDi sulla stessa tabella Persona.



Se in SQL vogliamo avere una tabella risultato con il cognome ed il nome delle persone accanto al cognome e nome dei genitori dobbiamo scrivere:

***SELECT** Tab1.Cognome, Tab1.Nome, Tab2.Cognome, Tab2.Nome*
***FROM** Persona **AS** Tab1, Persona **AS** Tab2*
WHERE** Tab1.CodPers1 = Tab2.CodPers2 **;

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

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

Tali operatori sono **UNION** (per l'operazione relazionale di **unione o** ∪), **INTERSECT** (per l'operazione relazionale di **intersezione o** ∩) e **MINUS** (per l'operazione relazionale di **differenza o** −).

Esempio: Consideriamo le seguenti relazioni
***Regista** (CodRegista, Cognome, Nome)*
***Attore** (CodAttore, Cognome, Nome)*

Per ottenere tutti i registi e tutti gli attori scriveremo
(SELECT** Cognome, Nome **FROM** Regista**)
UNION
(SELECT** Cognome, Nome **FROM** Attore**)

Per ottenere i registi che sono stati anche attori scriveremo
(SELECT** Cognome, Nome **FROM** Regista**)
INTERSECT
(SELECT** Cognome, Nome **FROM** Attore**)

Per ottenere i registi che sono non stati mai attori scriveremo
(SELECT** Cognome, Nome **FROM** Regista**)
MINUS
(SELECT** Cognome, Nome **FROM** Attore**)

*Naturalmente le due relazioni parziali delle **SELECT** devono essere compatibili per poter essere utilizzate come operandi degli operatori di intersezione, unione e differenza.*

Le interrogazioni parametriche

Pur non supportate dallo standard ANSI le interrogazioni parametriche sono molto utili quando si utilizza l'SQL in modalità stand-alone.
E' possibile par5ametrizzare una query in modo da scriverla una volta sola e sfruttarla per diversi valori del parametro. Per far ciò basta far precedere la query dalla seguente dichiarazione:

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

Esempio
PARAMETERS Nome, Data:DATE
SELECT *
FROM Cliente
WHERE NomeCliente = Nome AND DataNascita = Data;

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

Le funzioni di aggregazione

Il linguaggio SQL possiede alcune funzioni predefinite utilissime in molte circostanze in cui occorre effettuare conteggi, somme, calcoli, medie o altro ancora.
Tali funzioni si applicano ad una colonna di una tabella.
La loro sintassi è:

<FunzioneDiAggregazione> **([DISTINCT]** <NomeAttributo> **);**

dove la funzione di aggregazione può essere:
- l'istruzione **COUNT** che conteggia il **numero di elementi** della colonna specificata in <NomeAttributo>;
- l'istruzione **MIN** che restituisce il **valore minimo** della colonna specificata in <NomeAttributo>;
- l'istruzione **MAX** che restituisce il **valore massimo** della colonna specificata in <NomeAttributo>;
- l'istruzione **SUM** che restituisce la **somma degli elementi** della colonna specificata in <NomeAttributo>;
- l'istruzione **AVG** che restituisce la media aritmetica degli elementi della colonna specificata in <NomeAttributo>.

Ordinamenti

Finora non abbiamo fatto alcuna ipotesi **sull'ordine** in cui possono apparire le righe di una tabella risultato dell'istruzione **SELECT** (ossia di una query).
In SQL è possibile ordinare tali righe utilizzando alcune clausole che seguono l'istruzione SELECT

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

dove **ASC** sta per *ordine crescente* (ed è l'ordinamento di default) e **DESC** per *ordine decrescente*.
L'ordinamento viene eseguito dapprima sul primo attributo e poi a parità di ordinamento su di esso si ordina sulla base del secondo attributo e così via.

Esempio: per ordinare in ordine alfabetico decrescente i dipendenti che guadagnano più di 3000,00 euro scriveremo in SQL
***SELECT** **
***FROM** Dipendente*
***WHERE** Stipendio > 3000,00*
ORDER BY** Cognome **DESC** , Nome **DESC ;

Per ordinare in ordine alfabetico crescente i dipendenti scriveremo in SQL
***SELECT** **
***FROM** Dipendente*
ORDER BY** Cognome, Nome**;

Raggruppamenti

Le funzioni di aggregazione sono in genere abbinate alle clausole di raggruppamento la cui sintassi è la seguente:

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

Con l'aggiunta di questa clausola il significato della **SELECT** è il seguente:
- viene eseguito il prodotto delle tabelle presenti nella clausola **FROM**;
- su tale prodotto si effettua una restrizione in base alla clausola **WHERE**;
- la tabella risultante viene logicamente partizionata in gruppi di righe in base alla clausola **GROUP BY**. (due righe appartengono allo stesso gruppo se hanno gli stessi valori per gli attributi elencati nella clausola GROUP BY);
- tutti i gruppi che non soddisfano la clausola **HAVING** vengono eliminati.

Esempio: Per raggruppare i dipendenti in base al loro livello e conoscere lo stipendio medio per livello possiamo scrivere:
***SELECT** Livello, **AVG**(Stipendio)*
***FROM** Dipendente*
GROUP BY** Livello **;

Per raggrupparli in livelli solo per quelli maggiori del sesto scriveremo
***SELECT** Livello, **AVG**(Stipendio)*
***FROM** Dipendente*
***GROUP BY** Livello*
HAVING** Livello > 6 **;

Query annidate e subquery

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

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

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

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

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

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

Esempio di sottoquery che produce una tabella derivata
Consideriamo le seguenti relazioni relative all'utilizzo di laboratori da parte di una classe di studenti:
***Laboratorio** (CodLab, NumPosti, NomeLab)*
***Classe** (CodClasse, NumPosti)*
***Utilizza** (CodLab, CodClasse)*

E consideriamo la seguente interrogazione:
Q1: Vogliamo conoscere il nome dei laboratori utilizzati dalla classe "A45"

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

SUB-QUERY

Una sottoquery può essere composta a sua volta da altre query.
Si viene a creare così una struttura di **query annidate**.
Per eseguire sottoquery annidate si deve eseguire prima l'interrogazione più interna e poi eseguire quella più esterna fino ad arrivare alla query principale.

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

Esempio: consideriamo le seguenti relazioni
Regista (CodRegista, Cognome, Nome, Compenso)
Film (CodFilm, Titolo, Budget, Regia) con Regia FK su Regista (CodRegista)

Supponiamo di volere effettuare la seguente interrogazione
Q2**: **Mostrare il cognome ed il nome del regista del film "Via col vento"

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

analogo alla query

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

Sottointerrogazioni che producono valori appartenenti ad un insieme

Nella costruzione delle sottointerrogazioni è possibile utilizzare i seguenti predicati per effettuare ricerche sui valori di attributi che soddisfano proprietà di appartenenza a insiemi di valori.
I possibili predicati che possiamo utilizzare sono:
- **ANY** e **ALL**;
- **IN** e **NOT IN**;
- **EXISTS** e **NOT EXISTS**

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

```
SELECT <Lista Attributi>
FROM <ListaTabelle>
WHERE <Attributo> <Operatore Relazionale> ANY | ALL <SubQuery>
```

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

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

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

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

- **IN:** la condizione della clausola WHERE è vera se il valore dell'attributo <Attributo> **appartiene** all'insieme dei valori forniti dalla sottoquery < SubQuery >
(N.B. il predicato **IN** equivale a **= ANY);**
- **NOT IN:** la condizione della clausola WHERE è vera se il valore dell'attributo <Attributo> **non appartiene** all'insieme dei valori forniti dalla sottoquery < SubQuery >.
(N.B. il predicato **NOT IN** equivale a **< > ALL);**

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

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

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

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

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

Subquery nella FROM

E' possibile utilizzare una subquery anche nella clausola FROM, con questa sintassi:

SELECT ...
FROM (subquery) [AS] *nome* ...

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

Ad esempio:

```
SELECT sq.*, t2.c1
FROM (SELECT c1, c2, c3
      FROM t1
      WHERE c1 > 5) AS sq
LEFT JOIN t2 ON sq.c1 = t2.c1;
```

In questo caso l'output della subquery viene chiamato "sq" ed il riferimento è usato sia nella SELECT sia nella condizione di join.

Il valore NULL

Il linguaggio SQL prevede un particolare simbolo indicato con **NULL** che viene utilizzato per indicare diverse situazioni (ad esempio quando il valore esiste ma è sconosciuto oppure quando il valore non esiste).
Il valore del simbolo **NULL** assume un ruolo significativo nel risultato di una espressione logica o aritmetica.

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

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

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

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

VALORI	**NOT**
FALSE	TRUE
TRUE	FALSE
UNKNOWN	FALSE

Attenzione al significato di **NULL** in quanto l'espressione **WHERE Stipendio = NULL** è errata nella sintassi non essendo il valore **NULL** una costante, mentre l'espressione
WHERE Stipendio IS NULL è sintatticamente corretta**.**

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

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

ISTRUZIONE DEL DCL di SQL

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

Lo standard SQL non specifica in quale modo debba essere possibile creare, eliminare o modificare gli utenti di un database. La maggior parte dei DBMS a tale scopo implementa i comandi non standard CREATE USER (che crea un utente e specifica quali permessi deve avere) e DROP USER (che elimina un utente). Per la modifica, alcuni DBMS implementano ALTER USER, ma non è un comando molto diffuso, mentre è più diffuso SET PASSWORD.
In sintesi l'istruzione per creare un utente è:

CREATE USER <user>@<host> **[IDENTIFIED BY** <password>];

dove lo username per MYSQL assume la forma <user>@<host> e può essere eseguito solo da utenti che hanno il permesso giusto (privilegio di poter effettuare la CREATE USER).

L'istruzione per eliminare un utente è:

DROP USER <user>@<host>

Esempi:

```
CREATE USER davide@localhost;

CREATE USER davide@localhost IDENTIFIED BY 'pippo';

DROP USER davide@localhost;
```

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

Quando si parla di **sicurezza dei dati** occorre distinguere i seguenti aspetti:
- sicurezza da **guasti** *hardware* e *software*;
- sicurezza da **accessi** *non autorizzati: per proteggersi da questa eventualità è necessario:*
    - stabilire i **diritti di accesso**;
    - stabilire le **viste** ossia le modalità con le quali gli utenti possono vedere la base dei dati.
- sicurezza nelle **transazioni** per ottenere la quale occorre preservare **integrità** e **consistenza** dei dati.

N.B. Tralasciamo per il momento la sicurezza sui guasti hardware e software dei quali si parlerà approfonditamente quando dettaglieremo il DBMS

Diritti di accesso ai dati

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

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

```
GRANT <ElencoPrivilegi>
ON <NomeDB>.<NomeTabella>
TO <user>@<host>
IDENTIFIED BY<password>;
```

Come vedete la sintassi qui proposta è molto semplice, ma spiegamone comunque i singoli campi:

- **<ElencoPrivilegi>**: E' una lista di istruzioni di SQL che si vogliono permettere all'utente (CREATE, SELECT, UPDATE, DELETE, ALTER, EXECUTE, ecc..). Se si vuole dare all'utente permessi completi si può utilizzare la parola chiave **ALL**.
- **<NomeDB>**: E' il nome del database che stiamo prendendo in considerazione.
- **<NomeTabella>**: Inserendo il nome di una tabella, si fa riferimento solo ad essa. Per tutte le altre tabelle non varranno le regole che stiamo specificando. Se si vuole fare riferimento a tutte le tabella si può utilizzare il carattere asterisco (*).
- **<user>**: Specifica il nome dell'utente che vogliamo creare
- **<host>**: Specifica il/gli host da cui è ammessa la connessione
- **<password>**: Specifica la password (scritta tra apici singoli o doppi apici) associata all'utente che stiamo creando. La password va scritta "in chiaro". Se si desidera inserire la password in forma criptata tramite la funzione PASSWORD() di MySQL, si deve far precedere la stringa criptata dalla parola PASSWORD.

Tipicamente viene eseguita dall'utente *root*, mediante l'uso del comando GRANT.

Esempio di comandi a consolle MYSQL:

```
C:\>mysql -u root -p
Enter password: *******

mysql> GRANT ALL ON <NomeDB>.* TO
-> <user'>@<localhost> IDENTIFIED BY
-> <password>;
```

Per consentire la connessione da un server specifico

```
mysql> GRANT ALL ON <NomeDB>.* TO
-> <user>@<nome_server> IDENTIFIED BY
-> <password>;
```

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

```
REVOKE <ElencoPrivilegi>
FROM <NomeDB>.<NomeTabella>
TO <user>@<host> ;
```

Le viste

Una **vista** è una **relazione** che non è fisicamente memorizzata sulla base di dati (dove invece sono fisicamente memorizzate tutte le relazioni create attraverso l'istruzione di **CREATE TABLE**) ottenuta tramite un'operazione di **"mapping"** (mappatura) con le tabelle effettivamente memorizzate.

Dopo essere stata creata, una vista può essere **modificata** ed **interrogata**.

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

CREATE VIEW <NomeTabellaVista> **AS** <Query> **;**

Dove:
- <NomeTabellaVista> è il nome assegnato alla fittizia tabella della vista;
- <Query> è una normale query formulata con l'istruzione **SELECT**.

Come si intuisce il motivo per cui si creano le **viste** è quello di fornire ad un gruppo di utenti **una versione semplificata o parziale** di una realtà che può essere molto più complessa.
Categorie diverse di utenti possono interagire con la base di dati utilizzando il loro punto di vista e trascurando quelli degli altri.

N.B. Ogni modifica apportata su di una tabella vista si ripercuote sulla tabella dalla quale è stata tratta.

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

DROP VIEW <NomeTabellaVista> **;**

Sulle viste è possibile utilizzare l'istruzione **GRANT**.

SQL EMBEDDED

Finora abbiamo ipotizzato di utilizzare SQL in modalità **stand-alone** ossia abbiamo supposto che le istruzioni SQL venissero scritte attraverso una qualche interfaccia grafica e venissero eseguite dall'interprete SQL:
La modalità **embedded** si riferisce all'uso di istruzioni SQL all'interno di un linguaggio di programmazione detto **linguaggio di programmazione ospite**.
I questo caso oltre alle istruzioni SQL ed alle istruzioni proprie del linguaggio ospite, esiste un terzo tipo di istruzioni che permettono **l'integrazione ed il controllo del flusso** delle istruzioni SQL all'interno del linguaggio ospite.

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

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

Occorre quindi utilizzarli entrambi anche perché:
a) tutti gli attuali **DBMS** hanno **l'SQL** come **DML**;
b) lo standard **ANSI** ha definito estensioni particolari per molti linguaggi al fine di includere **l'SQL**.

Le istruzioni SQL per interfacciamento variano a seconda che si tratti di:
- linguaggio ospite **procedurale**;
- linguaggio ospite **ad oggetti.**

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

DEF: Un **cursore** è un **indicatore di posizione** che viene utilizzato per muoversi all'interno di una *tabella* risultato di una *query*.
Esso può essere visto come un **puntatore** ad una particolare riga di una tabella, riga che prende il nome di **riga corrente**.

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

SQL DINAMICO

La tecnica dell'**SQL dinamico o dynamic SQL** non prevede l'inserimento *statico* di istruzioni SQL nel codice del programma ospite ma la loro acquisizione *dinamica o a tempo di esecuzione* da qualsiasi dispositivo di input, oppure la loro generazione a programma.
Quindi le istruzioni SQL non sono note più *a tempo di compilazione* come avviene per **l'SQL embedded** ma vengono create o acquisiste *a tempo di esecuzione* dal programma ospite.

Con l'SQL dinamico è possibile mantenere indefiniti fino al momento dell'esecuzione del programma:
- il *testo* dell'istruzione SQL;
- il *numero* ed i *tipi* delle variabili ospiti;
- i *riferimenti* agli oggetti della base di dati.;

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

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

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

EXEC SQL EXECUTE IMMEDIATE <Query>

dove:
- <Query> è una stringa che rappresenta una query (composta nel programma ospite) oppure una variabile ospite di tipo stringa.
Questa modalità di esecuzione avviene per istruzioni SQL che non richiedono parametri né in ingresso né in uscita come i comandi di inserimento o cancellazione.

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

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

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

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

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

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

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