



## SQL

Annalisa Franco, Dario Maio  
Università di Bologna

# SQL: caratteristiche generali

- ❑ SQL (Structured Query Language) è il linguaggio **standard de facto** per DBMS relazionali, che riunisce in sé funzionalità di:
  - ❑ DDL = Data Definition Language;
  - ❑ DML = Data Manipulation Language;
  - ❑ DCL = Data Control Language.
- ❑ SQL è nato come un linguaggio **dichiarativo** (non-procedurale), ovvero **non specifica la sequenza di operazioni da compiere per ottenere il risultato**.
- ❑ SQL è “**relazionalmente completo**”, nel senso che ogni espressione dell'algebra relazionale può essere tradotta in SQL.
- ❑ Il modello dei dati di SQL è basato su **tabelle anziché relazioni**:
  - ❑ possono essere presenti **righe (tuple) duplicate**;
  - ❑ in alcuni casi l'ordine delle colonne (attributi) ha rilevanza;
  - ❑ ...il motivo è pragmatico (ossia legato a considerazioni sull'efficienza).
- ❑ SQL adotta la **logica a 3** valori introdotta con l'Algebra Relazionale.

# SQL: standard e dialetti

- ❑ Il processo di standardizzazione di SQL è iniziato nel 1986.
- ❑ Nel 1992 è stato definito lo standard **SQL-2** (o SQL-92) da parte dell'ISO (International Standards Organization), e dell'**ANSI** (American National Standards Institute), rispettivamente descritti nei documenti ISO/IEC 9075:1992 e ANSI X3.135-1992.
- ❑ Del 1999 è attivo lo standard **SQL:1999** che rende SQL un **linguaggio computazionalmente completo** (e quindi con istruzioni di controllo!) per il supporto di oggetti persistenti.
- ❑ Sono stati rilasciati altri standard nel 2003, 2006 e 2008. A seguito delle novità introdotte negli ultimi anni, **SQL si sta trasformando in un linguaggio sempre più procedurale**.
- ❑ Allo stato attuale **ogni sistema ha ancora un suo dialetto che:**
  - è compatibile (in larga parte) con SQL-2;
  - ha già elementi degli standard successivi;
  - ha anche costrutti non standard.

# Gli standard SQL

Year	Name	Comments
1986	SQL-86	First formalized by ANSI.
1989	SQL-89	Minor revision, adopted as FIPS 127-1.
1992	SQL-92	Major revision (ISO 9075), Entry Level SQL-92 adopted as FIPS 127-2.
1999	SQL:1999	Added regular expression matching, recursive queries, triggers, support for procedural and control-of-flow statements, non-scalar types, and some object-oriented features.
2003	SQL:2003	Introduced XML-related features, window functions, standardized sequences, and columns with auto-generated values (including identity-columns).
2006	SQL:2006	ISO/IEC 9075-14:2006 defines ways in which SQL can be used in conjunction with XML. It defines ways of importing and storing XML data in an SQL database, manipulating it within the database and publishing both XML and conventional SQL-data in XML form. In addition, it enables applications to integrate into their SQL code the use of XQuery, the XML Query Language published by the World Wide Web Consortium (W3C), to concurrently access ordinary SQL-data and XML documents.
2008	SQL:2008	Legalizes ORDER BY outside cursor definitions. Adds INSTEAD OF triggers. Adds the TRUNCATE statement.
2011	SQL:2011	Adds temporal data (PERIOD FOR) (more information at: Temporal database#History). Enhancements for window functions and FETCH clause.
2016	SQL:2016	Adds row pattern matching, polymorphic table functions, JSON.

# Data Definition Language (DDL)

- ❑ Il DDL di SQL permette di definire **scemi di relazioni** (o “**table**”, tabelle), modificarli ed eliminarli.
- ❑ Permette inoltre di specificare **vincoli**, sia a livello di tupla (o “**riga**”) sia a livello di tabella.
- ❑ Permette di definire nuovi **domini**, oltre a quelli predefiniti
  - ❑ Per vincoli e domini si può anche fare uso del DML (quindi inizialmente non è obbligatorio definirli completamente).
- ❑ Inoltre si possono definire **viste** (“**view**”), ovvero tabelle virtuali, e **indici**, per accedere efficientemente ai dati.

# Creazione ed eliminazione di tabelle

- Mediante l'istruzione **CREATE TABLE** si definisce lo schema di una tabella e se ne crea un'istanza vuota:

- per ogni attributo va specificato il **dominio**, un eventuale valore di default e eventuali **vincoli**;
- infine possono essere espressi altri vincoli a livello di tabella.

```
CREATE TABLE Impiegati (  
    CodImp      char(4)          PRIMARY KEY,          -- chiave primaria  
    CF          char(16)         NOT NULL UNIQUE,      -- chiave  
    Cognome     varchar(60)      NOT NULL,  
    Nome        varchar(30)      NOT NULL,  
    Sede        char(3)          REFERENCES Sedi(Sede), -- FK  
    Ruolo       char(20)         DEFAULT 'Programmatore',  
    Stipendio   int              CHECK (Stipendio > 0),  
    UNIQUE (Cognome, Nome)      -- chiave )
```

- Mediante l'istruzione **DROP TABLE** è possibile eliminare lo schema di una tabella (e conseguentemente la corrispondente istanza):  
**DROP TABLE** Impiegati

# Modifica di tabelle

- Mediante l'istruzione **ALTER TABLE** è possibile modificare lo schema di una tabella, in particolare:
  - aggiungendo attributi;
  - aggiungendo o rimuovendo vincoli.

```
ALTER TABLE Impiegati
```

```
ADD COLUMN Sesso char(1) CHECK (Sesso in ('M', 'F'))
```

```
ADD CONSTRAINT StipendioMax CHECK (Stipendio < 4000)
```

```
DROP CONSTRAINT StipendioPositivo
```

```
DROP UNIQUE (Cognome, Nome) ;
```

- Se si aggiunge un attributo con vincolo NOT NULL, si deve prevedere un valore di default che il sistema assegnerà automaticamente a tutte le tuple già presenti:

```
ADD COLUMN Istruzione char(10) NOT NULL DEFAULT 'Laurea'
```

# I domini

- In SQL sono utilizzabili 2 tipi di domini
  - **Domini elementari** (**predefiniti**):
    - **carattere**: singoli caratteri o stringhe, anche di lunghezza variabile;
    - **bit**: singoli booleani o stringhe;
    - **numerici**, esatti e approssimati;
    - **data, ora, intervalli di tempo.**
  - **Domini definiti dall'utente** (**semplici**): utilizzabili in definizioni di relazioni, anche con vincoli e valori di default. Si definiscono tramite l'istruzione:  

```
CREATE DOMAIN Voto AS SMALLINT  
DEFAULT NULL  
CHECK ( value >=18 AND value <= 30 )
```



# Vincoli (1)

## ➤ Valori di default e valori NULL:

- Per vietare la presenza di valori nulli, è sufficiente imporre il vincolo **NOT NULL**:

Cognome                      varchar(60)                      **NOT NULL**

- Per ogni attributo è inoltre possibile specificare un valore di default:

Ruolo                      char(20)                      **DEFAULT 'Programmatore'**

## ➤ Chiavi:

- La definizione di una chiave avviene esprimendo un vincolo **UNIQUE**, che si può specificare in linea, se la chiave consiste di un singolo attributo:

CF                      char(16)                      **UNIQUE**

- o dopo aver dichiarato tutti gli attributi, se la chiave consiste di uno o più attributi:

**UNIQUE (Cognome , Nome)**

- **PRIMARY KEY** definisce la chiave primaria:

CodImp   char(4)                      **PRIMARY KEY**

- *la specifica di una chiave primaria non è obbligatoria;*
- *si può specificare al massimo una chiave primaria per tabella;*
- *non è necessario specificare NOT NULL per gli attributi della primary key.*

# Vincoli (2)

- Chiavi straniere (“foreign key”)
  - ▣ La definizione di una foreign key avviene specificando un vincolo **FOREIGN KEY**, e indicando quale chiave viene referenziata;
  - ▣ le colonne di destinazione devono essere una chiave della tabella destinazione (non necessariamente la chiave primaria):  
**FOREIGN KEY** (Sede) **REFERENCES** Sedi (Sede)
- Vincoli generici (“check constraint”)
  - ▣ Mediante la clausola **CHECK** è possibile esprimere vincoli di tupla arbitrari, sfruttando tutto il potere espressivo di SQL. La sintassi è:  
**CHECK** (<condizione>)
  - ▣ Il vincolo è violato se esiste almeno una tupla che rende falsa la <condizione> (esclusi i valori NULL):  
Stipendio int **CHECK** (Stipendio > 0)
  - ▣ Se **CHECK** viene espresso a livello di tabella (anziché nella definizione dell'attributo) è possibile fare riferimento a più attributi della tabella stessa:  
**CHECK** (ImportoLordo = Netto + Ritenute)

# Politiche di “reazione”

- Anziché lasciare al programmatore il compito di garantire che siano rispettati i vincoli di integrità referenziale a fronte di cancellazioni e modifiche, si possono specificare opportune **politiche di reazione** in fase di definizione degli schemi.

```
CREATE TABLE Impiegati (  
  CodImp      char(4)          PRIMARY KEY,  
  Sede        char(3) ,  
  . . .  
  FOREIGN KEY Sede REFERENCES Sedi  
    ON DELETE CASCADE          -- cancellazione in cascata  
    ON UPDATE NO ACTION       -- modifiche non permesse
```

- Altre politiche: **SET NULL** e **SET DEFAULT**.

# DB di riferimento per gli esempi

## Impiegati

<u>CodImp</u>	Nome	Sede	Ruolo	Stipendio
E001	Rossi	S01	Analista	2000
E002	Verdi	S02	Sistemista	1500
E003	Bianchi	S01	Programmatore	1000
E004	Gialli	S03	Programmatore	1000
E005	Neri	S02	Analista	2500
E006	Grigi	S01	Sistemista	1100
E007	Violetti	S01	Programmatore	1000
E008	Aranci	S02	Programmatore	1200

## Sedi

<u>Sede</u>	Responsabile	Città
S01	Biondi	Milano
S02	Mori	Bologna
S03	Fulvi	Milano

## Progetti

<u>CodProg</u>	<u>Città</u>
P01	Milano
P01	Bologna
P02	Bologna

# L'istruzione SELECT

- È l'istruzione che permette di eseguire interrogazioni (*query*) sul DB.

```
SELECT [ALL | DISTINCT] [TOP (n) [PERCENT] [WITH TIES]]  
A1, A2, . . . , Am  
FROM R1, R2, . . . , Rn  
[WHERE <condizione>]  
[GROUP BY <listaAttributi>]  
[HAVING <condizione>]  
[ORDER BY <listaAttributi>]
```

ovvero:

- |  |                                    |
|--|------------------------------------|
| ■ <b>SELECT</b> (o <b>TARGET</b> ) <b>list</b> | (che cosa si vuole come risultato) |
| ■ <b>clausola FROM</b>                         | (da dove si prende)                |
| ■ <b>clausola WHERE</b>                        | (quali condizioni deve soddisfare) |
| ■ <b>clausola GROUP BY</b>                     | (le colonne su cui raggruppare)    |
| ■ <b>clausola HAVING</b>                       | (condizioni relative ai gruppi)    |
| ■ <b>clausola ORDER BY</b>                     | (ordinamento)                      |

**Il comando SELECT permette di realizzare le operazioni di *selezione*, *proiezione*, *join*, *raggruppamento* e *ordinamento*.**

# SELECT su singola tabella

*Codice, nome e ruolo dei dipendenti della sede S01*

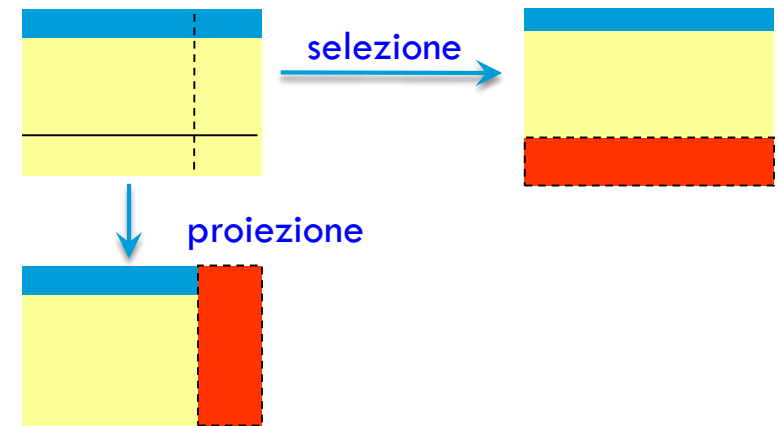
```
SELECT  CodImp, Nome, Ruolo
FROM    Impiegati
WHERE   Sede = 'S01'
```

CodImp	Nome	Ruolo
E001	Rossi	Analista
E003	Bianchi	Programmatore
E006	Grigi	Sistemista
E007	Violetti	Programmatore

- Si ottiene in questo modo:
  - ▣ la clausola FROM impone di accedere alla sola tabella IMP;
  - ▣ la clausola WHERE impone di selezionare solo le tuple per cui Sede= ' S01 ' ;
  - ▣ infine, si estraggono i valori degli attributi (o “colonne”) nella SELECT list.
- Equivale a  $\pi_{\text{CodImp, Nome, Ruolo}}(\sigma_{\text{Sede} = 'S01'}(\text{Impiegati}))$ .

# Selezione e proiezione in SQL

- Selezione e proiezione sono due operazioni “ortogonali”:
  - **Selezione:** realizza una decomposizione orizzontale includendo nel risultato solo le ennuple che soddisfano i requisiti; produce un risultato che:
    - Contiene tutti gli attributi dell'operando;
    - Contiene solo alcune ennuple dell'operando.
  - **Proiezione:** realizza una decomposizione verticale includendo nel risultato solo gli attributi richiesti; produce un risultato che:
    - ha solo una parte degli attributi dell'operando;
    - contiene un numero di ennuple pari quelle dell'operando.
  - **ATTENZIONE:** il risultato di una proiezione in SQL non è in generale una relazione poiché può contenere duplicati.



# SELECT senza proiezione

- Se si vogliono tutti gli attributi:

```
SELECT  CodImp, Nome, Sede, Ruolo, Stipendio  
FROM    Impiegati  
WHERE   Sede = 'S01'
```

si può abbreviare con:

```
SELECT  *  
FROM    Impiegati  
WHERE   Sede = 'S01'
```



# SELECT senza selezione (condizione)

- Con proiezione sugli attributi CodImp e Nome:

```
SELECT  CodImp, Nome  
FROM    Impiegati
```

- Se si vogliono tutte le tuple

```
SELECT  *  
FROM    Impiegati
```

restituisce l'intera estensione di Impiegati.

# Tabelle vs Relazioni: la clausola DISTINCT

- Il risultato di una query SQL può contenere **righe duplicate**:

```
SELECT Ruolo
FROM Impiegati
WHERE Sede = 'S01'
```

Ruolo
Analista
Programmatore
Sistemista
Programmatore

- Per eliminarle si usa l'opzione **DISTINCT** nella SELECT list:

```
SELECT DISTINCT Ruolo
FROM Impiegati
WHERE Sede = 'S01'
```

Ruolo
Analista
Programmatore
Sistemista

# La clausola TOP

(prende le prime n della Tabella)

- La clausola TOP specifica **quante righe** deve restituire, come risultato, la query.
- L'insieme di righe da restituire può essere specificato come **numero** o come **valore percentuale**.
- ➔ È possibile mantenere più record se hanno lo stesso valore per uno o più attributi (**WITH TIES**).

```
SELECT TOP (numero|percentuale) [PERCENT] [WITH TIES] A1,A2,...,Am  
FROM R1,R2,...,Rn
```

...

SQLServer,  
Access

➔ N.B. Non tutti i DBMS supportano la clausola TOP.

- Ciascun DBMS usa una propria sintassi:

MySQL

```
SELECT A1,A2,...,Am  
FROM R1,R2,...,Rn
```

...

```
LIMIT numero
```

```
SELECT A1,A2,...,Am  
FROM R1,R2,...,Rn
```

...

```
AND ROWNUM <= numero
```

Oracle

# La clausola TOP: esempi

```
SELECT CodImp, Nome
FROM Impiegati
WHERE Sede = 'S01'
```

CodImp	Nome
E001	Rossi
E003	Bianchi
E006	Grigi
E007	Violetti

```
SELECT TOP (2) CodImp, Nome
FROM Impiegati
WHERE Sede = 'S01'
```

CodImp	Nome
E001	Rossi
E003	Bianchi

```
SELECT TOP (20) PERCENT CodImp, Nome
FROM Impiegati
WHERE Sede = 'S01'
```

CodImp	Nome
E001	Rossi

# Espressioni complesse

- All'interno di un comando **SELECT** è possibile inserire espressioni booleane con operatori **AND** e **OR** e **NOT**:

```
SELECT  Nome
FROM    Impiegati
WHERE   Sede = 'S01'
OR      Ruolo = 'Programmatore'
```

```
SELECT  Nome
FROM    Impiegati
WHERE   Sede = 'S01'
AND     Ruolo = 'Programmatore'
```

Nome	Sede	Ruolo
Rossi	S01	Analista
Verdi	S02	Sistemista
Bianchi	S01	Programmatore
Gialli	S03	Programmatore
Neri	S02	Analista
Grigi	S01	Sistemista
Violetti	S01	Programmatore
Aranci	S02	Programmatore

# Operatore BETWEEN

- L'operatore **BETWEEN** permette di esprimere condizioni di appartenenza a un intervallo:

*Nome e stipendio degli impiegati che hanno uno stipendio compreso tra 1300 e 2000 Euro (estremi inclusi)*

```
SELECT Nome, Stipendio
FROM Impiegati
WHERE Stipendio BETWEEN 1300 AND 2000
```

Nome	Stipendio
Rossi	2000
Verdi	1500

- Lo stesso risultato si ottiene anche come segue:

```
SELECT Nome, Stipendio
FROM Impiegati
WHERE Stipendio >= 1300 AND Stipendio <= 2000
```

# Operatore IN

- L'operatore **IN** permette di esprimere condizioni di appartenenza a un insieme:

*Codici e sedi degli impiegati delle sedi S02 e S03*

```
SELECT  CodImp, Sede
FROM    Impiegati
WHERE   Sede IN ('S02', 'S03')
```

CodImp	Sede
E002	S02
E004	S03
E005	S02
E008	S02

- Lo stesso risultato si ottiene con gli operatori:

- ▣ **"=ANY"**

```
WHERE   Sede = ANY ('S02', 'S03')
```

- ▣ **"=" + "OR"**

```
WHERE   Sede = 'S02' OR Sede = 'S03'
```

# Operatore LIKE

- L'operatore **LIKE** permette di esprimere "pattern" su stringhe mediante le "wildcard" :
- ▣ **\_** (un carattere arbitrario)
  - ▣ **%** (una stringa arbitraria)

*Nomi degli impiegati che terminano con una 'i'  
e hanno una 'i' in seconda posizione*

```
SELECT Nome
FROM Impiegati
WHERE Nome LIKE '_i%i'
```

Nome
Bianchi
Gialli
Violetti



# Espressioni nella clausola SELECT

- La SELECT list può contenere non solo attributi, ma anche espressioni:

```
SELECT  CodImp, Stipendio*12
FROM    Impiegati
WHERE   Sede = 'S01'
```

CodImp	
E001	24000
E003	12000
E006	13200
E007	12000

- Le espressioni possono comprendere anche più attributi.
- Si noti che in questo caso la seconda colonna non ha un nome.

# Ridenominazione delle colonne

- A ogni elemento della SELECT list è possibile associare un nome a piacere:

```
SELECT CodImp AS Codice, Stipendio*12 AS StipendioAnnuo
FROM Impiegati
WHERE Sede = 'S01'
```

Codice	StipendioAnnuo
E001	24000
E003	12000
E006	13200
E007	12000

- La parola chiave **AS** può anche essere omessa:

```
SELECT CodImp Codice, ...
```

# Pseudonimi

- Per chiarezza ogni nome di colonna può essere scritto aggiungendo ad esso, come prefisso, il nome della tabella (obbligatorio in caso di ambiguità):

```
SELECT  Impiegati.CodImp AS Codice,  
        Impiegati.Stipendio*12 AS StipendioAnnuo  
FROM    Impiegati  
WHERE   Impiegati.Sede = 'S01'
```

...e si può anche usare uno pseudonimo (*alias*) in luogo del nome della tabella

```
SELECT  I.CodImp AS Codice,  
        I.Stipendio*12 AS StipendioAnnuo  
FROM    Impiegati I          -- oppure Impiegati AS I  
WHERE   I.Sede = 'S01'
```

# Valori nulli

- Il trattamento dei valori nulli si basa su quanto già visto in algebra relazionale, quindi la query:

```
SELECT  CodImp
FROM    Impiegati
WHERE   Stipendio > 1500
       OR Stipendio <= 1500
```

restituisce solo

CodImp
E001
E002
E003
E005
E007
E008

## Impiegati

CodImp	Sede	...	Stipendio
E001	S01		2000
E002	S02		1500
E003	S01		1000
E004	S03		NULL
E005	S02		2500
E006	S01		NULL
E007	S01		1000
E008	S02		1200

Non vengono considerati

# Logica a 3 valori in SQL

- Nel caso di espressioni complesse, SQL ricorre alla **logica a 3 valori**: vero (V), falso (F) e “sconosciuto” (?).

```
SELECT  CodImp, Sede, Stipendio
FROM    Impiegati
WHERE   (Sede = 'S03')
OR      (Stipendio > 1500)
```

CodImp	Sede	Stipendio
E001	S01	2000
E004	S03	NULL
E005	S02	2500

- Per verificare se un valore è NULL si usa l'operatore **IS**.
  - ▣ NOT (A **IS NULL**) si scrive anche A **IS NOT NULL**.

```
SELECT  CodImp
FROM    Impiegati
WHERE   Stipendio IS NULL
```

CodImp
E004
E006

# Ordinamento del risultato

- Per ordinare il risultato di una query secondo i valori di una o più colonne si introduce la clausola **ORDER BY**, e per ogni colonna si specifica se l'ordinamento è per valori “ascendenti” (**ASC**, il default) o “discendenti” (**DESC**)

```
SELECT    Nome, Stipendio
FROM      Impiegati
ORDER BY  Stipendio DESC
```

Nome	Stipendio
Neri	2500
Rossi	2000
Verdi	1500
Aranci	1200
Grigi	1100
Bianchi	1000
Gialli	1000
Violetti	1000

# Ordinamento e clausola TOP

- Può essere molto utile usare la clausola TOP in combinazione con ORDER BY.

*Nome dell'impiegato con ruolo 'Programmatore' che percepisce lo stipendio più basso*

```
SELECT TOP(1) Nome, Stipendio
FROM Impiegati
WHERE Ruolo = 'Programmatore'
ORDER BY Stipendio
```

(come default, si ha un ordinamento ascendente)

Nome	Stipendio
Bianchi	1000

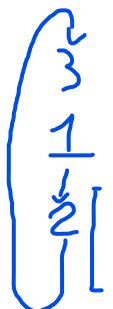
```
SELECT TOP(1) WITH TIES Nome, Stipendio
FROM Impiegati
WHERE Ruolo = 'Programmatore'
ORDER BY Stipendio
```

Nome	Stipendio
Bianchi	1000
Gialli	1000
Violetti	1000

N.B. WITH TIES si può usare solo in presenza di ORDER BY e i “pareggi” (TIES) si riferiscono alla combinazione degli attributi di ordinamento.

# Interrogazioni su più tabelle

## ➤ L'interrogazione



```
SELECT      I.Nome, I.Sede, S.Città
FROM        Impiegati I, Sedi S
WHERE       I.Sede = S.Sede
AND         I.Ruolo = 'Programmatore'
```

si interpreta come segue:

- ▣ si esegue il prodotto Cartesiano di Impiegati e Sedi;
- ▣ si applicano i predicati della clausola **WHERE**;
- ▣ si estraggono le colonne della **SELECT** list.

- Il predicato **I.Sede = S.Sede** è detto **predicato di join** in quanto stabilisce il criterio con cui le tuple di Impiegati e di Sedi devono essere combinate.



# Interrogazioni su più tabelle: risultato

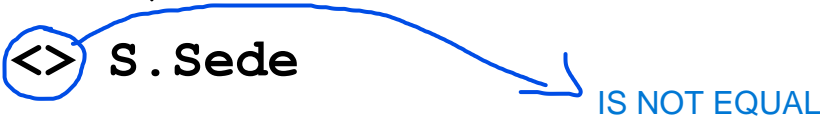
- Dopo avere applicato il predicato **I.Sede = S.Sede**:
- **celle in blu**: dopo avere applicato il predicato **I.Ruolo = 'Programmatore'** e la proiezione.

I.CodImp	I.Nome	I.Sede	I.Ruolo	I.Stipendio	S.Sede	S.Responsabile	S.Città
E001	Rossi	S01	Analista	2000	S01	Biondi	Milano
E002	Verdi	S02	Sistemista	1500	S02	Mori	Bologna
E003	Bianchi	S01	Programmatore	1000	S01	Biondi	Milano
E004	Gialli	S03	Programmatore	1000	S03	Fulvi	Milano
E005	Neri	S02	Analista	2500	S02	Mori	Bologna
E006	Grigi	S01	Sistemista	1100	S01	Biondi	Milano
E007	Violetti	S01	Programmatore	1000	S01	Biondi	Milano
E008	Aranci	S02	Programmatore	1200	S02	Mori	Bologna

# Ridenominazione del risultato

- Se la SELECT list contiene 2 o più colonne con lo stesso nome, è necessario operare una **ridenominazione** per ottenere un risultato in output con tutte le colonne dotate di intestazione:

```
SELECT      I.Sede AS SedeE001, S.Sede AS AltraSede
FROM        Impiegati I, Sedi S
WHERE       I.Sede <> S.Sede
AND         I.CodImp = 'E001'
```



IS NOT EQUAL

SedeE001	AltraSede
S01	S02
S01	S03

# Self Join

- L'uso di alias è forzato quando si deve eseguire un self-join:

*Chi sono i nonni di Anna?*

Genitori G1

Genitore	Figlio
Luca	Anna
Maria	Anna
Giorgio	Luca
Silvia	Maria
Enzo	Maria

Genitori G2

Genitore	Figlio
Luca	Anna
Maria	Anna
Giorgio	Luca
Silvia	Maria
Enzo	Maria

```
SELECT  G1.Genitore AS Nonno
FROM    Genitori G1, Genitori G2
WHERE   G1.Figlio = G2.Genitore
AND     G2.Figlio = 'Anna'
```

# Join espliciti

- Aniché scrivere i predicati di join nella clausola **WHERE**, è possibile “costruire” una **joined table** direttamente nella clausola FROM:

```
SELECT I.Nome, I.Sede, S.Città
FROM   Impiegati I JOIN Sedi S ON (I.Sede = S.Sede)
WHERE  I.Ruolo = 'Programmatore'
```

↳ condizione di join

in cui JOIN si può anche scrivere **INNER JOIN**.

- Altri tipi di join espliciti sono:

- **LEFT [OUTER] JOIN**
- **RIGHT [OUTER] JOIN**
- **FULL [OUTER] JOIN**
- **NATURAL JOIN**

# Outer join: esempi

*Per ciascuna sede visualizzare il nome del responsabile e il numero di Analisti che vi lavorano.*

**COUNT (I .CodImp)**

È importante applicare il count a un attributo di Impiegati per ottenere un valore 0 per le sedi senza analisti

```
SELECT S.Sede, S.Responsabile, COUNT(I.CodImp)
FROM Sedi S LEFT OUTER JOIN Impiegati I
ON (S.Sede = I.Sede) AND (I.Ruolo = 'Analista')
GROUP BY S.Sede, S.Responsabile
```

Predicato di join composto

Se in una sede non esistono Analisti  
gli attributi relativi all'impiegato  
saranno NULL

# Operatori insiemistici

- ➤ L'istruzione SELECT non permette di eseguire unione, intersezione e differenza di tabelle.
- ➤ Ciò che si può fare è combinare in modo opportuno i risultati di due istruzioni SELECT, mediante gli operatori:

UNION , INTERSECT , EXCEPT

↗ Differenza

- In tutti i casi gli elementi delle SELECT list devono avere **tipi compatibili** e gli stessi nomi se si vogliono colonne con un'intestazione definita.
- L'**ordine** degli elementi è importante (notazione posizionale).
- Il risultato è in ogni caso **privo di duplicati**, per mantenerli occorre aggiungere l'opzione ALL:

**UNION ALL , INTERSECT ALL , EXCEPT ALL**

# Operatori insiemistici: esempi (1)

R

A	B
1	a
1	a
2	a
2	b
2	c
3	b

S

C	B
1	a
1	b
2	a
2	c
3	c
4	d

```
SELECT A
FROM R
UNION
SELECT C
FROM S
```

1
2
3
4

```
SELECT A
FROM R
UNION
SELECT C AS A
FROM S
```

A
1
2
3
4

```
SELECT A,B
FROM R
UNION
SELECT B,C AS A
FROM S
```

**Non corretta!**

```
SELECT B
FROM R
UNION ALL
SELECT B
FROM S
```

B
a
a
a
b
c
b
a
b
c
c
d

# Operatori insiemistici: esempi (2)

Prendo Valori Uguali dell'attributo comune

R

A	B
1	a
1	a
2	a
2	b
2	c
3	b

```
SELECT B
FROM R
INTERSECT
SELECT B
FROM S
```

B
a
b
c

```
SELECT B
FROM S
EXCEPT
SELECT B
FROM R
```

B
d

È importante da  
quale tabella  
eseguo differenza

S

C	B
1	a
1	b
2	a
2	c
3	c
4	d

```
SELECT B
FROM R
INTERSECT ALL
SELECT B
FROM S
```

B
a
a
b
c

```
SELECT B
FROM R
EXCEPT ALL
SELECT B
FROM S
```

B
a
b



# Istruzioni di aggiornamento dei dati

- Le istruzioni che permettono di aggiornare il DB sono:

**INSERT** inserisce nuove tuple nel DB;

**DELETE** cancella tuple dal DB;

**UPDATE** modifica tuple del DB.

- **INSERT** può usare il risultato di una query per eseguire inserimenti multipli.
- **DELETE** e **UPDATE** possono fare uso di condizioni per specificare le tuple da cancellare o modificare.
- In ogni caso gli aggiornamenti riguardano una sola relazione.

# Inserimento di tuple: caso singolo

- È possibile inserire una nuova tupla specificandone i valori:

```
INSERT INTO Sedi (Sede, Responsabile, Città)
VALUES          ('S04', 'Bruni', 'Firenze')
```

- Deve esservi **corrispondenza** tra attributi e valori.
- La lista degli attributi si può omettere, nel qual caso vale l'ordine con cui sono stati definiti.
- Se la lista non include tutti gli attributi, i restanti assumono valore **NULL** (se ammesso) o il valore di **default** (se specificato):

```
INSERT INTO Sedi (Sede, Città) -- sede senza responsabile
VALUES          ('S04', 'Firenze')
```

# Inserimento di tuple: caso multiplo

- È possibile anche inserire le tuple che rappresentano il risultato di una query:

```
INSERT INTO SediBologna (SedeBO, Resp)
{
SELECT Sede, Responsabile
FROM Sedi
WHERE Città = 'Bologna'
```

- Gli schemi del risultato e della tabella in cui si inseriscono le tuple possono essere diversi, ma è necessario rispettare che i tipi delle colonne siano compatibili.

# Cancellazione e modifica di tuple

- L'istruzione **DELETE** può fare uso di una condizione per specificare le tuple da cancellare:

```
DELETE FROM   Sedi    -- elimina le sedi di Bologna
WHERE         Città = 'Bologna'
```

- Anche l'istruzione **UPDATE** può fare uso di una condizione, per specificare le tuple da modificare, e di espressioni per determinare i nuovi valori:

```
UPDATE   Sedi
SET      Responsabile = 'Bruni',
          Città = 'Firenze'
WHERE     Sede = 'S01'
```

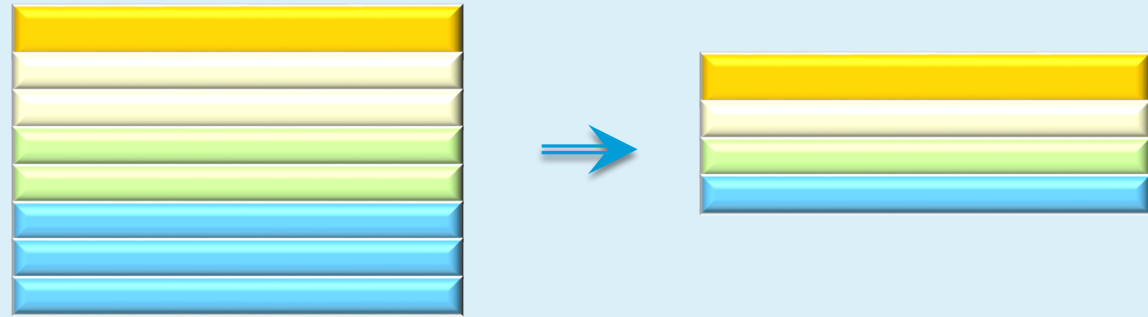
```
UPDATE Impiegati
SET      Stipendio = 1.1*Stipendio
WHERE     Ruolo = 'Programmatore'
```

# Data Manipulation Language (DML)

- Le istruzioni principali del DML di SQL sono:

<b>SELECT</b>	esegue interrogazioni ( <b>query</b> ) sul DB;
<b>INSERT</b>	inserisce nuove tuple nel DB;
<b>DELETE</b>	cancella tuple dal DB;
<b>UPDATE</b>	modifica tuple del DB.

- **INSERT** può usare il risultato di una query per eseguire inserimenti multipli.
- **DELETE** e **UPDATE** possono fare uso di condizioni per specificare le tuple da cancellare o modificare.



Il linguaggio SQL: raggruppamenti

# Informazioni di sintesi

- Quanto sinora visto permette di estrarre dal DB informazioni che si riferiscono a **singole tuple** (eventualmente ottenute mediante operazioni di join).

Esempio: il ruolo dell'impiegato 'E001', il responsabile della sede 'S02', il ruolo degli impiegati della sede 'S01', ecc.

- In molti casi è viceversa utile ottenere dal DB informazioni (di sintesi) che caratterizzano **“gruppi” di tuple**.

Esempio: il numero di programmatori della sede 'S01', la media degli stipendi a Bologna, ecc.

- A tale scopo SQL mette a disposizione due strumenti di base:
  - ▣ **funzioni aggregate**;
  - ▣ clausola **GROUP BY**.

# Funzioni aggregate (1)

- Lo standard SQL mette a disposizione una serie di **funzioni aggregate** (o “di colonna”):

- ▣ **MIN** minimo;
- ▣ **MAX** massimo;
- ▣ **SUM** somma;
- ▣ **AVG** media aritmetica;
- ▣ **STDEV** deviazione standard;
- ▣ **VARIANCE** varianza;
- ▣ **COUNT** contatore.

```
SELECT SUM(Stipendio) AS TotStipS01
FROM Impiegati
WHERE Sede = 'S01'
```

Ridenominazione  
Funzione Aggregata

TotStipS01
5100



# Funzioni aggregate (2)

- L'argomento di una funzione aggregata è una **qualunque espressione** che può figurare nella SELECT list (**ma NON UN'ALTRA FUNZIONE AGGREGATA!**)

```
SELECT SUM(Stipendio*12) AS TotStipAnnuoS01
FROM Impiegati
WHERE Sede = 'S01'
```

TotStipAnnuoS01
61200

- Tutte le funzioni, ad eccezione di **COUNT**, ignorano i valori nulli.
- Il risultato è NULL se tutti i valori sono NULL.
- L'opzione **DISTINCT** considera solo i valori distinti:

```
SELECT SUM(DISTINCT Stipendio)
FROM Impiegati
WHERE Sede = 'S01'
```

4100

# COUNT e valori nulli

Qua, vengono specificati gli Attributi da considerare della Tabella Risultato

- La forma **COUNT(\*)** conta le tuple del risultato; viceversa, specificando una colonna, si omettono quelle con valore nullo in tale colonna:

## Impiegati

CodImp	Sede	...	Stipendio
E001	S01		2000
E002	S02		1500
E003	S01		1000
E004	S03		NULL
E005	S02		2500
E006	S01		NULL
E007	S01		1000
E008	S02		1200

```
SELECT    COUNT (*) AS NumImpS01
FROM      Impiegati
WHERE     Sede = 'S01'
```

NumImpS01
-----------

4
---

```
SELECT    COUNT (Stipendio)
AS NumStipS01
FROM      Impiegati
WHERE     Sede = 'S01'
```

NumStipS01
------------

3
---

# Funzioni aggregate e tipo del risultato

- Per alcune funzioni aggregate, al fine di ottenere il risultato desiderato, è necessario operare un *casting* dell'argomento:

## Impiegati

...	Stipendio
	2000
	1500
	1000
	1000
	2500
	1100
	1000
	1200

```
SELECT    AVG(Stipendio) AS AvgStip
FROM      Impiegati      -- valore esatto 1412.5
```

AvgStip
1412

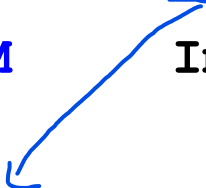
```
SELECT    AVG(CAST(Stipendio AS Decimal(6,2)))
          AS AvgStip
FROM      Impiegati
```

AvgStip
1412.50

# Clausola SELECT e funzioni aggregate

- Se si usano funzioni aggregate, la SELECT list **non può includere altri elementi** che non siano a loro volta funzioni aggregate:

```
SELECT  Nome, MIN(Stipendio)
FROM    Impiegati
```



**non va bene!** (perché Nome è un insieme di Valori, non un singolo valore)

(viceversa, **SELECT MIN(Stipendio), MAX(Stipendio) ..** è corretto)

- Il motivo è che **una funzione aggregata restituisce un singolo valore**, mentre **il riferimento a una colonna è in generale un insieme di valori** (eventualmente ripetuti).
- Nel caso specifico (chi sono gli impiegati con stipendio minimo?) è necessario ricorrere a un'altra soluzione, che vedremo più avanti.

# Funzioni aggregate e raggruppamento

- I valori di sintesi calcolati dalle funzioni aggregate si riferiscono a **tutte** le tuple che soddisfano le condizioni della clausola WHERE.
- Viceversa in molti casi è opportuno fornire i suddetti valori per **gruppi omogenei di tuple** (es: impiegati di una stessa sede).
- La clausola **GROUP BY** serve a definire tali gruppi, specificando una o più colonne (di raggruppamento) sulla base della/e quale/i le tuple sono raggruppate per valori uguali.

```
SELECT Sede, COUNT(*) AS NumProg
FROM Impiegati
WHERE Ruolo = 'Programmatore'
GROUP BY Sede
```

Sede	NumProg
S01	2
S03	1
S02	1

- **La SELECT list può includere solo le colonne di raggruppamento, ma non altre!**

# Come si ragiona con il GROUP BY

- Le tuple che soddisfano la clausola **WHERE**...
- ...sono raggruppate per valori uguali della/e colonna/e presenti nella clausola **GROUP BY**...
- ...e infine a ciascun gruppo si applica la funzione aggregata.

CodImp	Nome	Sede	Ruolo	Stipendio
E003	Bianchi	S01	Programmatore	1000
E004	Gialli	S03	Programmatore	1000
E007	Violetti	S01	Programmatore	1000
E008	Aranci	S02	Programmatore	1200

CodImp	Nome	Sede	Ruolo	Stipendio
E003	Bianchi	S01	Programmatore	1000
E007	Violetti	S01	Programmatore	1000
E004	Gialli	S03	Programmatore	1000
E008	Aranci	S02	Programmatore	1200

Sede	NumProg
S01	2
S03	1
S02	1

# GROUP BY: esempi

## 1) Per ogni ruolo, lo stipendio medio nelle sedi di Milano

```
SELECT  I.Ruolo, AVG(I.Stipendio) AS AvgStip
FROM    Impiegati I JOIN Sedi S ON (I.Sede = S.Sede)
WHERE   S.Città = 'Milano'
GROUP BY I.Ruolo
```

Ruolo	AvgStip
Analista	2000
Sistemista	1100
Programmatore	1000

## 2) Per ogni sede di Milano, lo stipendio medio

```
SELECT  I.Sede, AVG(I.Stipendio) AS AvgStip
FROM    Impiegati I JOIN Sedi S ON (I.Sede = S.Sede)
WHERE   S.Città = 'Milano'
GROUP BY I.Sede
```

Sede	AvgStip
S01	1275
S03	1000

## 3) Per ogni ruolo e sede di Milano, lo stipendio medio

```
SELECT I.Sede, I.Ruolo, AVG(I.Stipendio)
FROM Impiegati I JOIN Sedi S ON (I.Sede = S.Sede)
WHERE   S.Città = 'Milano'
GROUP BY I.Sede, I.Ruolo
```

Ruolo	Sede	
Analista	S01	2000
Sistemista	S01	1100
Programmatore	S01	1000
Programmatore	S03	1000

# Raggruppamento e proiezione

- Quando la SELECT list include solo le colonne di raggruppamento, il risultato è equivalente a ciò che si otterrebbe omettendo il **GROUP BY** e rimuovendo i duplicati con l'opzione **DISTINCT**.

```
SELECT    Sede
FROM      Impiegati
GROUP BY  Sede
```

Sede
S01
S02
S03

equivale pertanto a:

```
SELECT    DISTINCT Sede
FROM      Impiegati
```



# Condizioni sui gruppi

- Oltre a poter formare dei gruppi, è anche possibile **selezionare alcuni gruppi sulla base di loro proprietà “complessive”**:

```
SELECT Sede, COUNT (*) AS NumImp
FROM Impiegati
GROUP BY Sede
HAVING COUNT (*) > 2
```

Sede	NumImp
S01	4
S02	3

HAVING si utilizza solo se si usa il GROUP BY

- La clausola **HAVING** ha per i gruppi una funzione simile a quella che la clausola WHERE ha per le tuple (**attenzione a non confonderle!**).

# Tipi di condizioni sui gruppi

- Nella clausola HAVING si possono avere due tipi di predicati:
  - ▣ predicati che fanno uso di **funzioni aggregate** (es. **COUNT (\*) > 2**);
  - ▣ predicati che si riferiscono alle **colonne di raggruppamento**; questi ultimi si possono anche inserire nella clausola WHERE.

```
SELECT  Sede, COUNT (*) AS NumImp
FROM    Impiegati
GROUP BY Sede
HAVING  Sede <> 'S01'
```

Sede	NumImp
S02	3
S03	1

equivale a:

```
SELECT  Sede, COUNT (*) AS NumImp
FROM    Impiegati
WHERE   Sede <> 'S01'
GROUP BY Sede
```

# Un esempio completo

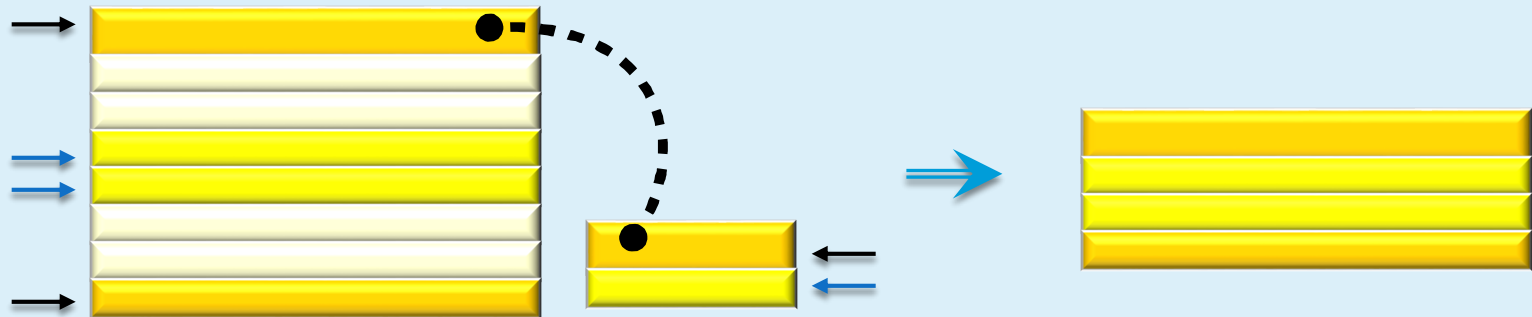
Per ogni sede di Bologna in cui il numero di impiegati è almeno 3, si vuole conoscere il valor medio degli stipendi, ordinando il risultato per valori decrescenti di stipendio medio e quindi per sede

```
SELECT    I.Sede, AVG(Stipendio) AS AvgStipendio
FROM      Impiegati I, Sedi S
WHERE     I.Sede = S.Sede
AND       S.Città = 'Bologna'
GROUP BY  I.Sede
HAVING    COUNT(*) >= 3
ORDER BY  AvgStipendio DESC, I.Sede
```



L'ordine delle clausole è sempre come nell'esempio.

Si ricorda che il GROUP BY non implica alcun ordinamento del risultato.



Il linguaggio SQL: query innestate

# Subquery

- Oltre alla forma “flat” vista sinora, in SQL è anche possibile esprimere **condizioni che si basano sul risultato di altre interrogazioni** (subquery, o query innestate o query nidificate):

```
SELECT    CodImp    -- impiegati delle sedi di Milano
FROM      Impiegati
WHERE     Sede IN (SELECT    Sede
                   FROM      Sedi
                   WHERE      Città = 'Milano')
```

Sede
S01
S03

- La subquery restituisce l'insieme di sedi ('S01','S03'), e quindi il predicato nella clausola WHERE esterna equivale a:

```
WHERE     Sede IN ('S01','S03')
```

# Subquery scalari

- Gli operatori di confronto `=`, `<`,... si possono usare solo se la subquery restituisce non più di una tupla (subquery “**scalare**”):

```
SELECT CodImp    -- impiegati con stipendio minimo
FROM Impiegati
WHERE Stipendio = (SELECT MIN(Stipendio)
                  FROM Impiegati)
```

MinStip
1000

- La presenza di vincoli può essere sfruttata a tale scopo:

```
SELECT Responsabile
FROM Sedi
WHERE Sede = (SELECT Sede -- al massimo una sede
              FROM Impiegati
              WHERE CodImp = 'E001')
```

Sede
S01

# Subquery: caso generale

- Se la subquery può restituire più di un valore si devono usare le forme:

- ▣ **<op> ANY**: la relazione <op> vale per almeno uno dei valori;
- ▣ **<op> ALL**: la relazione <op> vale per tutti i valori.

operatore di confronto

**SELECT**  
**FROM**  
**WHERE**

Responsabile

Sedi

Sede

= ANY

uguale ad almeno un valore

(**SELECT**  
**FROM**  
**WHERE**

Sede

Impiegati

Stipendio > 1500)

Quando ho subquery ed operatori di confronto, valuto i valori di un attributo uno per volta

**SELECT**  
**FROM**  
**WHERE**

CodImp

Impiegati

Stipendio

-- impiegati con stipendio minimo

<= ALL

(**SELECT**  
**FROM**

Stipendio  
Impiegati)

- La forma = ANY equivale a IN.

minore o uguale a tutti i valori

# Subquery: livelli multipli di innestamento

- Una subquery può fare uso a sua volta di altre subquery. Il risultato si può ottenere risolvendo a partire dal blocco più interno:

```
SELECT CodImp
FROM Impiegati
WHERE Sede IN (SELECT Sede
                FROM Sedi
                WHERE Città NOT IN (SELECT Città
                                    FROM Prog
                                    WHERE CodProg = 'P02'))
```

- **Attenzione a non sbagliare quando ci sono negazioni!** Nell'esempio, i due blocchi interni NON sono equivalenti a:

```
WHERE Sede IN (SELECT Sede
                FROM Sedi, Prog
                WHERE Sedi.Città <> Prog.Città
                AND   Prog.CodProg = 'P02')
```



# Subquery: quantificatore esistenziale

- Mediante **EXISTS** (SELECT \* ...) è possibile verificare se il risultato di una subquery restituisce almeno una tupla:

```
SELECT Sede  
FROM Sedi S
```

```
WHERE EXISTS (SELECT *  
              FROM Impiegati  
              WHERE Ruolo = 'Programmatore')
```

Seleziono quella Sede se  
la SubQuery restituisce  
almeno una tupla

- Facendo uso di **NOT EXISTS** il predicato è vero se la subquery non restituisce alcuna tupla.
- In entrambi i casi la cosa non è molto “interessante” in quanto il risultato della subquery ~~è sempre lo stesso, ovvero~~ non dipende dalla specifica tupla del blocco esterno.

# Subquery correlate

- Se la subquery fa riferimento a “variabili” definite in un blocco esterno, allora si dice che è **correlata**:

```
SELECT Sede -- sedi con almeno un programmatore
FROM   Sedi S
WHERE  EXISTS (SELECT *
               FROM   Impiegati
               WHERE   Ruolo = 'Programmatore'
               AND     Sede = S.Sede)
```

↳ Adesso il risultato della query innestata dipende dalla sede specifica, e la semantica quindi diventa:

per ogni tupla del blocco esterno, considera il valore di S.Sede e risolvi la query innestata.

# Subquery: “unnesting” (1)

- È spesso possibile ricondursi a una forma “piatta”, ma la cosa non è sempre così ovvia. Ad esempio, l'interrogazione precedente si può anche scrivere:

```
SELECT    DISTINCT Sede
FROM      Sedi S, Impiegati I
WHERE     S.Sede = I.Sede
AND       I.Ruolo = 'Programmatore'
```

- Si noti la presenza del **DISTINCT**.
- La forma innestata è “più procedurale” di quella piatta e, a seconda dei casi, può risultare più semplice da derivare.
- Si ricordi comunque che in una subquery non si possono usare operatori insiemistici (UNION, INTERSECT e EXCEPT) e che una subquery può comparire solo come operando destro in un predicato.

# Subquery: “unnesting” (2)

- Con la negazione le cose tendono a complicarsi. Ad esempio, per trovare le sedi senza programmatori, nella forma innestata basta sostituire **NOT EXISTS** a **EXISTS**,

```
SELECT Sede          -- sedi senza programmatori
FROM Sedi S
WHERE NOT EXISTS (SELECT *
                  FROM Impiegati
                  WHERE Ruolo = 'Programmatore'
                  AND Sede = S.Sede)
```

Sede
S04

ma la forma piatta:

```
SELECT DISTINCT Sede
FROM Sedi S JOIN Impiegati I ON (S.Sede = I.Sede)
WHERE I.Ruolo <> 'Programmatore'
```

Sede
S01
S02
S03
S04

restituisce le sedi in cui **lavora almeno un impiegato con ruolo diverso da un programmatore**.

# Subquery: “unnesting” (3)

- Una soluzione “piatta” corretta (ma complessa) è:

```
SELECT DISTINCT Sede
FROM Sedi S LEFT OUTER JOIN Impiegati I ON
    (S.Sede = I.Sede) AND (I.Ruolo = 'Programmatore')
WHERE I.CodImp IS NULL
```

Sede
S04

- ❑ il join restituisce come risultato le sedi con relativi impiegati nel ruolo programmatore comprese le sedi senza programmatori;
- ❑ la clausola WHERE seleziona solo queste ultime.

- È facile sbagliare. Ad esempio la seguente query non è corretta:

```
SELECT DISTINCT Sede
FROM Sedi S LEFT OUTER JOIN Impiegati I ON (S.Sede = I.Sede)
WHERE I.Ruolo = 'Programmatore'
AND I.CodImp IS NULL
```

Sede
------

- ❑ il join restituisce come risultato le sedi con relativi impiegati (di qualsiasi ruolo) comprese le sedi senza impiegati;
- ❑ la clausola WHERE non è mai soddisfatta!

# Un esempio complesso

*Sede che ha il numero maggiore di impiegati con ruolo 'Programmatore'*

```
SELECT Sede, COUNT(CodImp) AS NumProg
FROM Impiegati I1
WHERE I1.Ruolo = 'Programmatore'
GROUP BY I1.Sede
HAVING COUNT(CodImp) >= ALL
      (SELECT COUNT(CodImp)
       FROM Impiegati I2
       WHERE I2.Ruolo = 'Programmatore'
       GROUP BY I2.Sede)
```

Sede	NumProg
S01	2

In alternativa...

```
SELECT TOP(1) WITH TIES Sede, COUNT(CodImp)
FROM Impiegati
WHERE Ruolo = 'Programmatore'
GROUP BY Sede
ORDER BY COUNT(CodImp) DESC
```

**N.B.** molte query innestate non possono essere semplificate usando la clausola TOP (es. *impiegati che hanno uno stipendio superiore a quello medio per il relativo ruolo*).

# Subquery: come eseguire la divisione

- Con le subquery è possibile eseguire la **divisione** relazionale.

1. Inizio con una Sede
2. Ciclo per tutti i Ruoli disponibili da verificare per quella sede

**“Sedi in cui sono presenti tutti i ruoli”**  
equivale a  
**“Sedi in cui non esiste un ruolo non presente”**

Sede/Ruolo  
= Sedi in cui  
ho tutti i ruoli

```
SELECT Sede  
FROM Sedi S  
WHERE NOT EXISTS
```

Sede
S01
S02

```
(SELECT Ruolo  
FROM Impiegati I1  
WHERE NOT EXISTS  
(SELECT *  
FROM Impiegati I2  
WHERE S.Sede = I2.Sede  
AND I1.Ruolo = I2.Ruolo))
```

Se non ci sono ruoli, che  
non sono in quella sede,  
allora Seleziono la Sede

Si sta essenzialmente  
cercando se per una  
determinata sede S.Sede,  
ci sono impiegati I2 che  
lavorano in quella sede.  
Si cerca di verificare se  
esiste un impiegato I2 nella  
sede S.Sede che ha  
esattamente lo stesso ruolo  
che si sta cercando  
(I1.Ruolo).

- Il blocco più interno è valutato per ogni combinazione di **sede** e di **ruolo**. passati dalle 2 query prec.
- Il blocco intermedio funge da **“divisore”** (interessa I1.Ruolo). (ciò che divide)
- Data una sede S, se in S manca un ruolo la subquery più interna non restituisce nulla;
  - ➔ la subquery intermedia restituisce almeno una tupla;
  - ➔ la clausola WHERE non è soddisfatta per S.

# La divisione: una formulazione alternativa

- L'espressione “Sedi in cui sono presenti tutti i ruoli” equivale anche a “Sedi per cui il numero di ruoli distinti è uguale al numero totale di ruoli presenti”.

```
SELECT Sede FROM Impiegati I
GROUP BY Sede
HAVING COUNT (DISTINCT Ruolo) =
(SELECT COUNT (DISTINCT Ruolo) FROM Impiegati)
```

Sede	NRuoli
S01	3
S02	3
S03	1

TotRuoli
3

Ogni sede ha un TOT di ruoli distinti che per essere Selezionata deve essere uguale al Numero TOT di ruoli disponibili

- il blocco più interno conta il numero di ruoli distinti;
- il blocco esterno raggruppa gli impiegati per sedi e conta il numero di ruoli distinti su ciascuna sede...
- ...selezionando solo le sedi per cui tale numero è pari al totale.

Sede
S01
S02



# La divisione: il semplice conteggio non è sempre possibile

- L'espressione "Sedi in cui sono presenti tutti i ruoli della sede 'S03' " **NON** equivale a "Sedi per cui il numero di ruoli distinti è uguale al numero di ruoli presenti nella sede 'S03' ".
- Occorre imporre un vincolo: solo ruoli presenti in 'S03'.

```
SELECT Sede FROM Impiegati I
WHERE Ruolo IN (SELECT Ruolo
                FROM Impiegati I2
                WHERE Sede = 'S03')
```

GROUP BY Sede

```
HAVING COUNT (DISTINCT Ruolo) =
(SELECT COUNT (DISTINCT Ruolo)
 FROM Impiegati
 WHERE Sede = 'S03')
```

Sede
S01
S02
S03

invece, senza condizione...

Sede
S03
S04

# Subquery: aggiornamento dei dati

- Le subquery si possono efficacemente usare per aggiornare i dati di una tabella sulla base di criteri che dipendono dal contenuto di altre tabelle:

```
DELETE FROM Impiegati -- elimina gli impiegati di Bologna
WHERE Sede IN (SELECT Sede
                FROM Sedi
                WHERE Città = 'Bologna')
```

```
UPDATE Impiegati -- aumenta lo stipendio agli impiegati
SET Stipendio = 1.1*Stipendio
WHERE Sede IN (SELECT S.Sede -- delle sedi di 'P02'
                FROM Sede S, Prog P
                WHERE S.Città = P.Città
                AND P.CodProg = 'P02')
```

# Subquery e CHECK

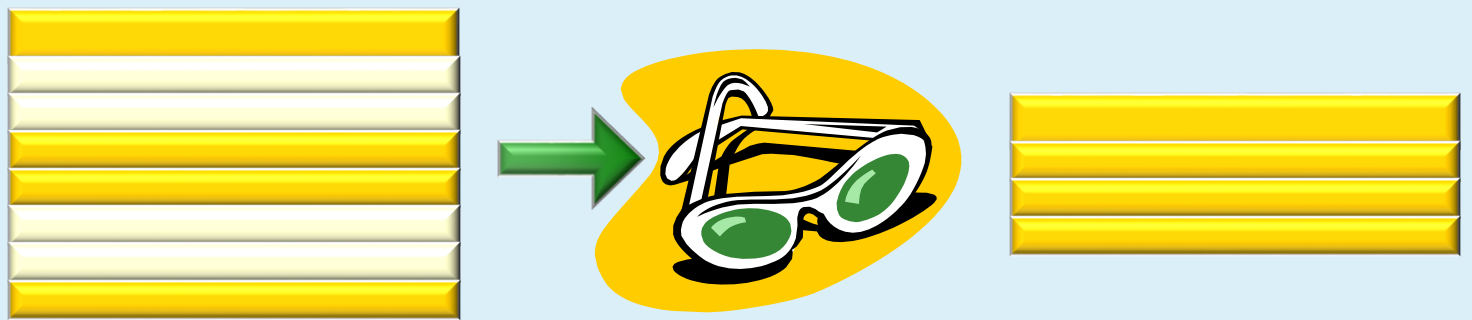
- Facendo uso di subquery nella clausola **CHECK** è possibile esprimere vincoli arbitrariamente complessi. Ad esempio:

*Ogni sede deve avere almeno due programmatori*

```
... -- quando si crea la TABLE Sedi
CHECK (2 <= (SELECT COUNT(*) FROM Impiegati I
            WHERE I.Sede = Sede -- correlazione
            AND I.Ruolo = 'Programmatore'))
```

Supponendo di avere due tabelle **ImpBO** e **ImpMI** e di volere che uno stesso codice (**CodImp**) non sia presente in entrambe le tabelle:

```
... -- quando si crea la TABLE ImpBO
CHECK (NOT EXISTS (SELECT * FROM ImpMI
                  WHERE ImpMI.CodImp = CodImp))
```



Il linguaggio SQL: viste e tabelle derivate

# Definizione di viste

La vista esegue dinamicamente la query sottostante quando vogliamo accedere alla vista

- Mediante l'istruzione **CREATE VIEW** si definisce una **vista**, ovvero una “**tabella virtuale**”.
- Le **tuple della vista** sono il risultato di una query che viene **valutata dinamicamente** ogni volta che si fa riferimento alla vista.

```
CREATE VIEW ProgSedi (CodProg, CodSede)
AS      SELECT P.CodProg, S.Sede
        FROM   Prog P, Sedi S
        WHERE  P.Città = S.Città
```

ProgSedi

CodProg	CodSede
P01	S01
P01	S03
P01	S02
P02	S02

```
SELECT *
FROM   ProgSedi
WHERE  CodProg = 'P01'
```

CodProg	CodSede
P01	S01
P01	S03
P01	S02

# Uso delle viste

- Le viste possono essere create a vari scopi, tra i quali si ricordano i seguenti:
  - ▣ permettere agli utenti di avere una **visione personalizzata del DB** che in parte astragga dalla struttura logica del DB stesso;
  - ▣ far fronte a **modifiche dello schema logico** che comporterebbero una ricompilazione dei programmi applicativi;
  - ▣ **semplificare la scrittura di query complesse**.
- Inoltre le viste possono essere usate come **meccanismo per il controllo degli accessi**, fornendo a ogni classe di utenti gli opportuni privilegi.
- Si noti che nella definizione di una vista si possono referenziare anche altre viste.

# Indipendenza logica tramite VIEW

- A titolo esemplificativo si consideri un DB che contiene la tabella  
**EsamiBD (Matr, Cognome, Nome, DataProva, Voto)**
- Per evitare di ripetere i dati anagrafici, si decide di modificare lo schema del DB sostituendo alla tabella **EsamiBD** le due seguenti:

**StudentiBD (Matr, Cognome, Nome)**

**ProveBD (Matr, DataProva, Voto)**

- È possibile ripristinare la “**visione originale**” in questo modo:

```
CREATE VIEW EsamiBD (Matr, Cognome, Nome, DataProva, Voto)  
AS          SELECT S.*, P.DataProva, P.Voto  
            FROM    StudentiBD S, ProveBD P  
            WHERE   S.Matr = P.Matr
```

# Query complesse che usano VIEW

- Un “classico” esempio di uso delle viste si ha nella scrittura di query di raggruppamento in cui si vogliono confrontare i risultati della funzione aggregata:

*La sede che ha il massimo numero di impiegati*

- La soluzione senza viste è:

```
SELECT    I.Sede
FROM      Impiegati I
GROUP BY  I.Sede
HAVING    COUNT (*) >= ALL (SELECT    COUNT (*)
                             FROM      Impiegati I1
                             GROUP BY  I1.Sede)
```



# Query complesse che usano VIEW

- La soluzione con viste è:

```
CREATE VIEW NumImp (Sede, NImp)
AS
    SELECT    Sede, COUNT (*)
    FROM      Impiegati
    GROUP BY  Sede
```

```
SELECT Sede
FROM    NumImp
WHERE   NImp = (SELECT MAX (NImp)
                FROM    NumImp)
```

NumImp

Sede	NImp
S01	4
S02	3
S03	1

che permette di trovare “il **MAX dei COUNT(\*)**”, cosa che, si ricorda, **non si può fare direttamente scrivendo MAX(COUNT(\*)**).

# Aggiornamento di viste

- Le viste possono essere utilizzate per le interrogazioni come se fossero tabelle del DB, ma **per le operazioni di aggiornamento ci sono dei limiti.**

```
CREATE VIEW NumImp (Sede, NImp)
AS      SELECT      Sede, COUNT (*)
        FROM        Impiegati
        GROUP BY    Sede
```

NumImp

Sede	NImp
S01	4
S02	3
S03	1

```
UPDATE NumImp
SET      NImp = NImp + 1
WHERE    Sede = 'S03'
```

- Che cosa significa? **Non si può fare!**

# Aggiornabilità di viste (1)

- Una vista è di fatto una funzione che calcola un risultato  $y$  a partire da un'istanza di database  $r$ ,  $y = V(r)$ .
- L'aggiornamento di una vista, che trasforma  $y$  in  $y'$ , può essere eseguito **se e solo se** è univocamente definita la nuova istanza  $r'$  tale che  $y' = V(r')$ , e ciò corrisponde a dire che **la vista è "invertibile", ossia  $r' = V^{-1}(y')$** .
- Data la complessità del problema, di fatto ogni DBMS pone dei limiti sulle tipologie di viste che possono essere aggiornate.
- Le **più comuni restrizioni** riguardano la non aggiornabilità di viste in cui **il blocco più esterno** della query di definizione contiene:
  - ▣ GROUP BY;
  - ▣ funzioni aggregate;
  - ▣ DISTINCT;
  - ▣ join (espliciti o impliciti).

# Aggiornabilità di viste (2)

- La precisazione che è **il blocco più esterno** della query di definizione che non deve contenere, ad es., dei join ha importanti conseguenze.
- Ad esempio, la vista:

```
CREATE VIEW ImpBO (CodImp, Nome, Sede, Ruolo, Stipendio)
AS      SELECT I.*
        FROM Impiegati I JOIN Sedi S ON (I.Sede = S.Sede)
        WHERE  S.Città = 'Bologna'
```

**non è aggiornabile** mentre lo è la seguente, di fatto equivalente alla prima:

```
CREATE VIEW ImpBO (CodImp, Nome, Sede, Ruolo, Stipendio)
AS SELECT I.*
    FROM  Impiegati I
    WHERE I.Sede IN (SELECT S.Sede FROM Sedi S
                    WHERE  S.Città = 'Bologna')
```

# Viste con CHECK OPTION

- Per le viste aggiornabili si presenta un nuovo problema. Si consideri il seguente inserimento nella vista **ImpBO**:

```
INSERT INTO ImpBO (CodImp, Nome, Sede, Ruolo, Stipendio)
VALUES ('E009', 'Azzurri', 'S03', 'Analista', 1800)
```

Anche se l'inserimento viene eseguito con successo, la riga inserita non rispetta la condizione della vista

in cui il valore di Sede ('S03') non rispetta la specifica della vista. Ciò comporta che una successiva query su **ImpBO** non restituirebbe la tupla appena inserita (!?).

- Per evitare situazioni di questo tipo, all'atto della creazione di una vista si può specificare, facendola seguire alla query che definisce la vista, la clausola **WITH CHECK OPTION**, che garantisce che ogni tupla inserita nella vista sia anche restituita dalla vista stessa.

La clausola WITH CHECK OPTION garantisce che ogni riga inserita o aggiornata tramite la vista soddisfi le condizioni della vista stessa. Se provi a inserire una riga che non rispetta queste condizioni, l'operazione fallirà

# Tipi di CHECK OPTION

Se V1 è definita in termini di V2, e si specifica WITH CHECK OPTION, il DBMS per default controlla che la riga inserita soddisfi sia le condizioni di V1 sia quelle di V2.

V1 non è definita direttamente su una tabella, ma usa V2 come base

- Se la vista V1 è definita in termini di un'altra vista V2, e si specifica la clausola **WITH CHECK OPTION**, il DBMS verifica che la nuova tupla t inserita soddisfi sia la definizione di V1 sia quella di V2, indipendentemente dal fatto che V2 sia stata a sua volta definita **WITH CHECK OPTION**.
- Questo comportamento di default, che è equivalente a definire V1 **WITH CASCADED CHECK OPTION** si può alterare definendo V1 **WITH LOCAL CHECK OPTION**
- In modalità **LOCAL**, il DBMS verifica solo che t soddisfi la specifica di V1 e quelle di **tutte e sole** le viste da cui V1 dipende per cui è stata specificata la clausola **WITH CHECK OPTION**.

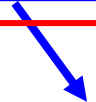
Se specifichi WITH LOCAL CHECK OPTION, il DBMS controllerà solo che la riga soddisfi le condizioni di V1 e quelle delle viste direttamente collegate a V1 che hanno CHECK OPTION. In questo caso, non si verifica necessariamente che la riga soddisfi tutte le condizioni delle viste sottostanti come in V2, a meno che queste non abbiano CHECK OPTION.

# Table expressions

- Tra le caratteristiche più interessanti di SQL vi è la possibilità di usare all'interno della *select list* o della clausola FROM una *subquery* che definisce “dinamicamente” una tabella derivata, e che qui viene anche detta “table expression”.

*Per ogni sede, lo stipendio massimo e quanti impiegati lo percepiscono*

```
SELECT SM.Sede, SM.MaxStip, COUNT(*) AS NumImpMaxStip
FROM Impiegati I, (SELECT Sede, MAX(Stipendio)
                   FROM Impiegati
                   GROUP BY Sede) AS SM(Sede, MaxStip)
WHERE I.Sede = SM.Sede
AND I.Stipendio = SM.MaxStip
GROUP BY SM.Sede, SM.MaxStip
```



SM

Sede	MaxStip
S01	2000
S02	2500
S03	1000

# Table expressions correlate (1)

- Le table expressions possono essere utilizzate anche nella select list e in tal caso possono anche essere correlate ad altre tabelle presenti nella clausola FROM. Ad esempio:

*Per ogni sede, la somma degli stipendi pagati agli analisti*

```
SELECT S.Sede, (SELECT SUM(Stipendio) FROM Impiegati I
                WHERE I.Sede = S.Sede
                AND I.Ruolo = 'Analista') AS TotStip
FROM   Sedi S
```

oppure...



# Table expressions correlate (2)

*Per ogni sede, la somma degli stipendi pagati agli analisti*

```
SELECT S.Sede, Stipendi.TotStip
FROM   Sedi S, (SELECT I.Sede, SUM(Stipendio)
                FROM Impiegati I
                WHERE I.Ruolo = 'Analista'
                GROUP BY Sede) AS Stipendi(Sede, TotStip)
WHERE  S.Sede = Stipendi.Sede
```

# Limiti delle table expressions

- Si consideri la query:

*la sede in cui la somma degli stipendi è massima*

- La soluzione con table expressions è:

```
SELECT Sede
FROM (SELECT Sede, SUM(Stipendio) AS TotStip
      FROM Impiegati
      GROUP BY Sede) AS SediStip
WHERE TotStip = (SELECT MAX(TotStip)
                FROM
                (SELECT Sede, SUM(Stipendio) AS TotStip
                  FROM Impiegati
                  GROUP BY Sede) AS SediStip2)
```

- Benché la query sia corretta, non è sfruttato il fatto che le due table expressions sono identiche, e ciò porta a una **valutazione inefficiente** e a una **formulazione poco leggibile**. (bisogna riscrivere la Table anche all'interno di una subquery)

# Common table expressions

- L'idea alla base delle “common table expressions” è definire una “**vista temporanea**” che può essere usata in una query come se fosse a tutti gli effetti una VIEW:

```
WITH SediStip(Sede, TotStip)
AS (SELECT Sede, SUM(Stipendio)
    FROM Impiegati
    GROUP BY Sede)
```

common table  
expression  
(validità all'interno  
di una singola  
query)

```
SELECT Sede
FROM SediStip
WHERE TotStip = (SELECT MAX(TotStip)
                 FROM SediStip)
```

# WITH e interrogazioni ricorsive (1)

- Si consideri la **tabella** **Genitori (Figlio, Genitore)** e la query

*Trova tutti gli **antenati** (genitori, nonni, bisnonni,...) di Anna*

- La query è **ricorsiva** e pertanto **non è esprimibile in algebra relazionale**, in quanto **richiede un numero di (self-)join non noto a priori**.
- La formulazione mediante common table expressions definisce la vista temporanea (ricorsiva) **Antenati (Persona, Avo)** facendo l'unione di:
  - ▣ una “subquery base” non ricorsiva (che inizializza **Antenati** con le tuple di **Genitori**);
  - ▣ una “subquery ricorsiva” che a ogni iterazione aggiunge ad **Antenati** le tuple che risultano dal join tra **Genitori** e **Antenati**.

**Genitori**

Figlio	Genitore
Anna	Luca
Luca	Maria
Luca	Giorgio
Giorgio	Lucia

**Antenati**

Persona	Avo
Anna	Luca
Luca	Maria
Luca	Giorgio
Giorgio	Lucia

+

**Antenati**

Persona	Avo
Anna	Maria
Anna	Giorgio
Luca	Lucia

+

**Antenati**

Persona	Avo
Anna	Lucia

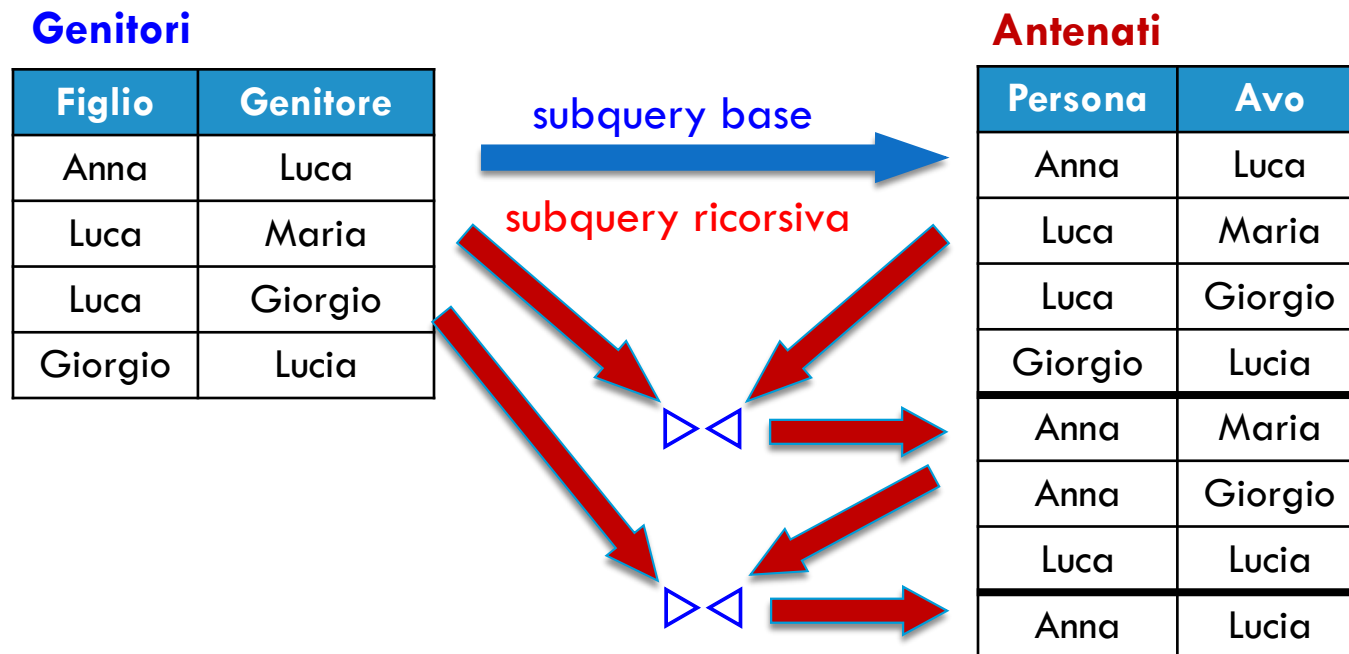
# WITH e interrogazioni ricorsive (2)

```
WITH Antenati (Persona, Avo)
AS (
    (SELECT Figlio, Genitore           -- subquery base
     FROM Genitori)
    UNION ALL                          -- sempre UNION ALL!
    (SELECT G.Figlio, A.Avo            -- subquery ricorsiva
     FROM Genitori G, Antenati A
     WHERE G.Genitore = A.Persona)
)
```

```
SELECT Avo
FROM Antenati
WHERE Persona = 'Anna'
```

# WITH e interrogazioni ricorsive (3)

- Per comprendere meglio come funziona la valutazione di una query ricorsiva, e come “ci si ferma”, si tenga presente che a ogni iterazione il DBMS aggiunge ad **Antenati** le tuple che risultano dal join tra **Genitori** e le sole tuple aggiunte ad **Antenati** al passo precedente:



# Uso delle common table expressions

- Una common table expression è soggetta ad alcune limitazioni d'uso, in particolare si può usare solo nel **blocco più esterno di un SELECT** (anche se usato per creare VIEW o eseguire INSERT).
- Per le subquery ricorsive vi sono alcune restrizioni, tra cui va ricordato che **non si possono usare funzioni aggregate, GROUP BY e SELECT DISTINCT**.

# Domande?

---

