



# Basi di dati

**Maurizio Lenzerini**

***Dipartimento di Informatica e Sistemistica “Antonio Ruberti”  
Università di Roma “La Sapienza”***

Anno Accademico 2020/2021

<http://www.dis.uniroma1.it/~lenzerini/?q=node/44>



# SQL

- originariamente "**S**tructured **Q**uery **L**anguage", ora "nome proprio"
- è un linguaggio con varie funzionalità, che contiene:
  - il DDL (Data Definition Language)
  - il DML (Data Manipulation Language)
- ne esistono varie versioni
- analizziamo gli aspetti essenziali non i dettagli
- un po' di storia:
  - prima proposta **SEQUEL** (IBM Research, 1974);
  - prime implementazioni in SQL/DS (IBM) e Oracle (1981);
  - dal 1983 ca., "standard di fatto"
  - standard (1986, poi 1989, poi **1992**, 1999, e infine 2003):  
recepito solo in parte



# SQL-92

- è un linguaggio ricco e complesso
- ancora nessun sistema mette a disposizione tutte le funzionalità del linguaggio
- 3 livelli di aderenza allo standard:
  - **Entry SQL**: abbastanza simile a SQL-89
  - **Intermediate SQL**: caratteristiche più importanti per le esigenze del mercato; supportato dai DBMS commerciali
  - **Full SQL**: funzioni avanzate, in via di inclusione nei sistemi
- i sistemi offrono funzionalità non standard
  - incompatibilità tra sistemi
  - incompatibilità con i nuovi standard (es. trigger in SQL:1999)
- Nuovi standard conservano le caratteristiche di base di SQL-92:
  - **SQL:1999** aggiunge alcune funzionalità orientate agli oggetti
  - **SQL:2003** aggiunge supporto per dati XML



# Utilizzo di un DBMS basato su SQL

- Un DBMS basato su SQL consente di gestire basi di dati relazionali; dal punto di vista sistemistico è un **server**
- Quando ci si connette ad un DBMS basato su SQL, si deve indicare, implicitamente o esplicitamente, su quale basi di dati si vuole operare
- Se si vuole operare su una base di dati non ancora esistente, si utilizzerà un meccanismo messo a disposizione dal server per la sua creazione
- Coerentemente con la filosofia del modello relazionale, una base di dati in SQL è caratterizzata dallo **schema** (livello intensionale) e da una **istanza** (quella corrente -- livello estensionale)
- In più, una base di dati SQL è caratterizzata da un insieme di **meta-dati** (il catalogo – vedi dopo)



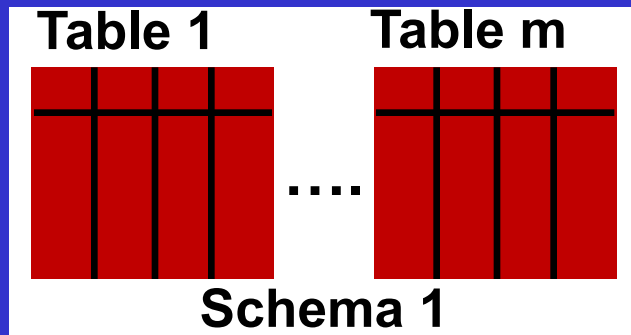
## SQL e modello relazionale

- **Attenzione:** una tabella in SQL è definita come un multiinsieme di ennuple
- In particolare, se una tabella non ha una primary key o un insieme di attributi definiti come unique, allora potranno comparire due ennuple uguali nella tabella; ne segue che una tabella SQL **non** è in generale una relazione
- Se invece una tabella ha una primary key o un insieme di attributi definiti come unique, allora non potranno mai comparire nella tabella due ennuple uguali e quindi in questo caso una tabella è una relazione; per questo, è consigliabile definire almeno una primary key per ogni tabella

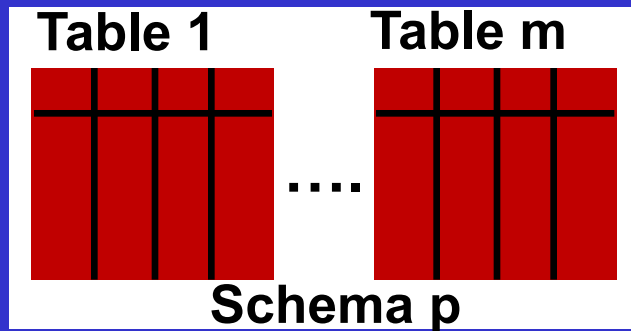
# Basi di dati, schemi e tabelle in PostgreSQL

## Installazione server PostgreSQL

### Database 1

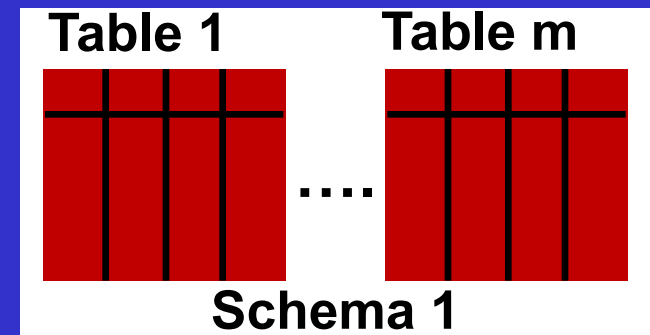


⋮

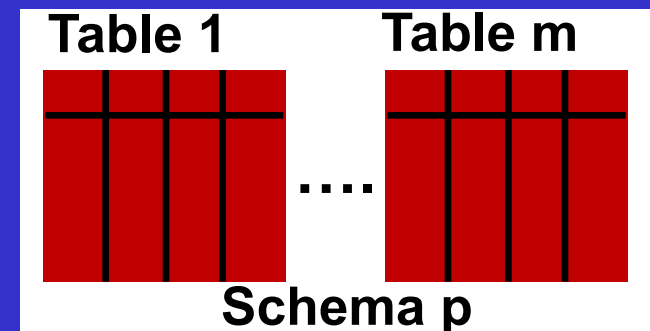


.....

### Database n



⋮





## 3. Il Linguaggio SQL

### 3.1 Definizione dei dati

1. **interrogazioni semplici**
2. definizione dei dati
3. manipolazione dei dati
4. interrogazioni complesse
5. ulteriori aspetti



# Istruzione select (versione base)

- L'istruzione di interrogazione in SQL è

**select**

che definisce una interrogazione, e **restituisce il risultato in forma di tabella**

```
select  Attributo ... Attributo  
from    Tabella ... Tabella  
[where  Condizione]
```

- le tre parti vengono di solito chiamate
  - **target list**
  - **clausola from**
  - **clausola where**





## maternita

madre	figlio
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

## paternita

padre	figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

## persone

nome	eta	reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	30
Luigi	50	40
Franco	60	20
Olga	30	41
Sergio	85	35
Luisa	75	87

in tutte le slides che seguono  
assumiamo che persone diverse  
abbiano nomi diversi



# Selezione e proiezione

Nome e reddito delle persone con meno di 30 anni

$\text{PROJ}_{\text{nome, reddito}}(\text{SEL}_{\text{eta} < 30}(\text{persone}))$

```
select persone.nome, persone.reddito  
from   persone  
where  persone.eta < 30
```

nome	reddito
Andrea	21
Aldo	15
Filippo	30



## Convenzioni sui nomi

- Per evitare ambiguità, ogni nome di attributo è composto da *NomeTabella.NomeAttributo*
- Quando l'ambiguità non sussiste, si può omettere la parte *NomeTabella.*

```
select persone.nome, persone.reddito  
from persone  
where persone.eta < 30
```

si può scrivere come:

```
select nome, reddito  
from persone  
where eta < 30
```



## SELECT, abbreviazioni

```
select persone.nome, persone.reddito  
from   persone  
where  persone.eta < 30
```

si può scrivere anche:

```
select p.nome as nome, p.reddito as reddito  
from   persone as p  
where  p.eta < 30
```

o anche:

```
select p.nome nome, p.reddito reddito  
from   persone p  
where  p.eta < 30
```



# Proiezione, attenzione

Cognome e filiale di tutti gli impiegati

**impiegati**

matricola	cognome	filiale	stipendio
7309	Neri	Napoli	55
5998	Neri	Milano	64
9553	Rossi	Roma	44
5698	Rossi	Roma	64

**PROJ** cognome, filiale **(impiegati)**



## Proiezione, attenzione

```
select cognome,  
       filiale  
from impiegati
```

cognome	filiale
Neri	Napoli
Neri	Milano
Rossi	Roma
Rossi	Roma

```
select distinct cognome,  
       filiale  
from impiegati
```

cognome	filiale
Neri	Napoli
Neri	Milano
Rossi	Roma



## SELECT, uso di “as”

“**as**” nella lista degli attributi serve a specificare esplicitamente un nome per un attributo del risultato. Quando per un attributo manca tale ridenominazione, il nome nel risultato sarà uguale a quello che compare nella tabella.

*Esempio:*

```
select nome as nomePersone, reddito as salario
from   persone
where  eta < 30
```

restituisce come risultato una relazione con due attributi, il primo di nome **nomePersone** ed il secondo di nome **salario**

```
select nome, reddito
from   persone
where  eta < 30
```

restituisce come risultato una relazione con due attributi, il primo di nome **nome** ed il secondo di nome **reddito**



# Esercizio 1

Calcolare la tabella ottenuta dalla tabella **persone** selezionando solo le persone con reddito tra 20 e 30 aggiungendo un attributo che ha, in ogni ennupla, lo stesso valore dell'attributo **reddito**

Mostrare il risultato dell'interrogazione.

**persone**

<b>nome</b>	<b>eta</b>	<b>reddito</b>
-------------	------------	----------------





## Soluzione esercizio 1

```
select nome, eta, reddito,  
       reddito as ancoraReddito  
from   persone  
where  reddito >= 20 and reddito <= 30
```

nome	eta	reddito	ancoraReddito
Andrea	27	21	21
Filippo	26	30	30
Franco	60	20	20



## Selezione, senza proiezione

Nome, età e reddito delle persone con meno di 30 anni

**SEL<sub>eta<30</sub>(persone)**

```
select *  
from   persone  
where  eta < 30
```

è un'abbreviazione per:

```
select nome, eta, reddito  
from   persone  
where  eta < 30
```



tutti gli  
attributi



## SELECT con asterisco

Data una relazione **R** sugli attributi **A**, **B**, **C**

```
select  *  
from    R  
where   cond
```

equivale a

```
select  A, B, C  
from    R  
where   cond
```



## Proiezione, senza selezione

Nome e reddito di tutte le persone

**PROJ**<sub>nome, reddito</sub>(persone)

```
select nome, reddito  
from persone
```

è un'abbreviazione per:

```
select p.nome, p.reddito  
from persone p  
where true
```



## Espressioni nella target list

```
select reddito/2 as redditoSemestrale  
from   persone  
where  nome = 'Luigi'
```

## Condizione complessa nella clausola “where”

```
select *  
from   persone  
where  reddito > 25  
       and (eta < 30 or eta > 60)
```



## Condizione “LIKE”

Le persone che hanno un nome che inizia per 'A', ha 'd' come terza lettera e può continuare con altri caratteri

```
select *  
from   persone  
where  nome like 'A_d%'
```



## Gestione dei valori nulli

Gli impiegati la cui età è o potrebbe essere maggiore di 40

**SEL** `eta > 40 OR eta IS NULL` (**impiegati**)

```
select *  
from   impiegati  
where  eta > 40 or eta is null
```



## Esercizio 2

Calcolare la tabella ottenuta dalla tabella **impiegati** selezionando solo quelli delle filiali di Roma e Milano, proiettando i dati sull'attributo **stipendio**, ed aggiungendo un attributo che ha, in ogni ennupla, il valore doppio dell'attributo **stipendio**

Mostrare il risultato dell'interrogazione

<b>impiegati</b>	<b>matricola</b>	<b>cognome</b>	<b>filiale</b>	<b>stipendio</b>
------------------	------------------	----------------	----------------	------------------





## Soluzione esercizio 2

```
select stipendio,  
       stipendio*2 as stipendiobis  
from   impiegati  
where  filiale = 'Milano' or  
       filiale = 'Roma'
```

stipendio	stipendiobis
64	128
44	88
64	128



## Selezione, proiezione e join

- Istruzioni **select** con una sola relazione nella clausola **from** permettono di realizzare:
  - selezioni
  - proiezioni
  - ridenominazioni
- I **join** (e i prodotti cartesiani) si realizzano indicando due o più relazioni nella clausola **from**



# SQL e algebra relazionale, 1

Date le relazioni:  $R1(A1,A2)$  e  $R2(A3,A4)$

la semantica della query

```
select R1.A1, R2.A4
from   R1, R2
where  R1.A2 = R2.A3
```

si può descrivere in termini di

- prodotto cartesiano (**from**)
- selezione (**where**)
- proiezione (**select**)

Attenzione: questo non significa che il sistema calcola davvero il prodotto cartesiano!



## SQL e algebra relazionale, 2

Date le relazioni:  $R1(A1, A2)$  e  $R2(A3, A4)$

```
select R1.A1, R2.A4
from   R1, R2
where  R1.A2 = R2.A3
```

corrisponde a:

$PROJ_{A1, A4} (SEL_{A2=A3} (R1 \text{ JOIN } R2))$

*Siccome R1 e R2 non hanno attributi in comune, il join naturale corrisponde al prodotto cartesiano*



## SQL e algebra relazionale, 3

Possono essere necessarie ridenominazioni

- nella target list (come nell'algebra relazionale)
- nel prodotto cartesiano (in particolare quando occorre riferirsi due volte alla stessa tabella)

```
select X.A1 as B1, ...  
from    R1 as X, R2 as Y, R1 as Z  
where   X.A2 = Y.A3 and ...
```

che si scrive anche

```
select X.A1 B1, ...  
from    R1 X, R2 Y, R1 Z  
where   X.A2 = Y.A3 and ...
```



## SQL e algebra relazionale: esempio

```
select X.A1 as B1, Y.A4 as B2
from    R1 X, R2 Y, R1 Z
where   X.A2 = Y.A3 and Y.A4 = Z.A1
```

```
RENB1,B2←A1,A4 (
    PROJA1,A4 (SELA2 = A3 and A4 = C1 (
        R1 JOIN R2 JOIN RENC1,C2 ← A1,A2 (R1))))
```



# SQL: esecuzione delle interrogazioni

- Le espressioni SQL sono dichiarative e noi ne stiamo vedendo la semantica
- In pratica, i DBMS eseguono le operazioni in modo efficiente, ad esempio:
  - eseguono le selezioni al più presto
  - se possibile, eseguono join e **non** prodotti cartesiani
- La capacità dei DBMS di "**ottimizzare**" le interrogazioni, rende (di solito) non necessario preoccuparsi dell'efficienza quando si specifica un'interrogazione
- È perciò più importante preoccuparsi della chiarezza (anche perché così è più difficile sbagliare ...)



## maternita

madre	figlio
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

## paternita

padre	figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

## persone

nome	eta	reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	30
Luigi	50	40
Franco	60	20
Olga	30	41
Sergio	85	35
Luisa	75	87

in tutte le slides che seguono  
assumiamo che persone diverse  
abbiano nomi diversi





## Assunzioni

Nelle slide che seguono, facciamo queste assunzioni:

- Persone diverse hanno nomi diversi
- Ogni figlio ha un solo padre
- Ogni figlio ha una sola madre
- Se non esplicitamente detto, non ci preoccupiamo di eliminare i duplicati nel risultato delle query



## Esercizio 3: selezione, proiezione e join

I padri di persone che guadagnano più di venti milioni  
(senza ripetizioni nel risultato)

Esprimere la query sia in algebra relazionale sia in SQL



## Esercizio 3: soluzione

I padri di persone che guadagnano più di venti milioni  
(senza ripetizioni nel risultato)

PROJ<sub>padre</sub>(paternita JOIN<sub>figlio=nome</sub> SEL<sub>reddito>20</sub> (persone))

```
select distinct paternita.padre
from   persone, paternita
where  paternita.figlio = persone.nome
       and persone.reddito > 20
```



## Esercizio 4: join

Padre e madre di ogni persona

Esprimere la query sia in algebra relazionale sia in SQL.



## Esercizio 4: soluzione

Padre e madre di ogni persona

In algebra relazionale si calcola mediante il **join naturale**.

paternita JOIN maternita

In SQL:

```
select paternita.figlio, padre, madre
from   maternita, paternita
where  paternita.figlio = maternita.figlio
```



## Esercizio 4: soluzione

Se avessimo inteso la domanda come: padre e madre di ogni persona che appare nella tabella “persona”, allora avremmo dovuto usare un join in più:

In algebra:

$$\text{PROJ}_{\text{figlio, padre, madre}} ((\text{paternita JOIN maternita}) \\ \text{JOIN}_{\text{figlio=nome}} \text{persone})$$

In SQL:

```
select paternita.figlio, padre, madre
from   maternita, paternita, persone
where  paternita.figlio = maternita.figlio
       and paternita.figlio = persone.nome
```



## Esercizio 5: join e altre operazioni

Le persone che guadagnano più dei rispettivi padri, mostrando nome, reddito e reddito del padre

Esprimere la query sia in algebra relazionale sia in SQL



## Esercizio 5: soluzione

Le persone che guadagnano più dei rispettivi padri; mostrare nome, reddito e reddito del padre

$$\text{PROJ}_{\text{nome, reddito, RP}} (\text{SEL}_{\text{reddito} > \text{RP}} (\text{REN}_{\text{NP, EP, RP} \leftarrow \text{nome, eta, reddito}} (\text{persone})) \text{ JOIN}_{\text{NP=padre}} (\text{paternita JOIN}_{\text{figlio = nome}} \text{ persone})))$$

```
select    f.nome, f.reddito, p.reddito
from      persone p, paternita t, persone f
where     p.nome = t.padre and
          t.figlio = f.nome and
          f.reddito > p.reddito
```





## SELECT, con ridenominazione del risultato

Le persone che guadagnano più dei rispettivi padri; mostrare nome, reddito e reddito del padre

```
select figlio, f.reddito as reddito,  
       p.reddito as redditoPadre  
from   persone p, paternita t, persone f  
where  p.nome = t.padre and  
       t.figlio = f.nome and  
       f.reddito > p.reddito
```



## SELECT con join esplicito, sintassi

```
select ...  
from Tabella { join Tabella on CondDiJoin },  
...  
[ where AltraCondizione ]
```

È l'operatore SQL corrispondente allo theta-join



## Join esplicito

Padre e madre di ogni persona

```
select paternita.figlio, padre, madre  
from    maternita, paternita  
where   paternita.figlio = maternita.figlio
```

```
select madre, paternita.figlio, padre  
from    maternita join paternita on  
        paternita.figlio = maternita.figlio
```

join  
esplicito



## Esercizio 6: join esplicito

Le persone che guadagnano più dei rispettivi padri, mostrando nome, reddito e reddito del padre

Esprimere la query in SQL usando il join esplicito



## SELECT con join esplicito, esempio

Le persone che guadagnano più dei rispettivi padri, mostrando nome, reddito e reddito del padre

```
select f.nome, f.reddito, p.reddito
from   persone p, paternita t, persone f
where  p.nome = t.padre and
       t.figlio = f.nome and
       f.reddito > p.reddito
```

```
select f.nome, f.reddito, p.reddito
from   persone p join paternita t on p.nome=t.padre
       join persone f on t.figlio=f.nome
where  f.reddito > p.reddito
```



# SELECT con join esplicito: ridenominazione

Ricordiamo che il risultato del join è una tabella che ha come attributi l'unione degli attributi dei due operandi.

```
select f.nome, f.reddito, p.reddito
from   persone p join paternita t on p.nome=t.padre
       join persone f on t.figlio=f.nome
where  f.reddito > p.reddito
```

Per esempio, nella query mostrata sopra, il join esplicito che compare nella clausola «from» dà come risultato una tabella con gli attributi: p.nome, p.eta, p.reddito, t.padre, t.figlio, f.nome, f.eta, ed f.reddito.

Si noti che il risultato del join si può anche usare come una delle tabelle nella lista della clausola «from», ma in questo occorre racchiudere il join esplicito tra parentesi tonde (e gli si può anche assegnare un alias, con la solita notazione):

```
select f.nome, f.reddito, p.reddito
from   (persone p join paternita t on p.nome=t.padre),
       persone f
where  f.reddito > p.reddito and t.figlio = f.nome
```



## Ulteriore estensione: join naturale (meno diffuso)

**PROJ**<sub>figlio,padre,madre</sub>(**paternita** JOIN<sub>figlio←nome</sub> **REN**<sub>nome←figlio</sub>(**maternita**))

In algebra: paternita JOIN maternita

In SQL (con join esplicito):  

```
select paternita.figlio, padre, madre  
from  maternita join paternita on  
      paternita.figlio = maternita.figlio
```

In SQL (con natural join):  

```
select paternita.figlio, padre, madre  
from maternita natural join paternita
```



## Join esterno: "outer join"

Padre e, se nota, madre di ogni persona

```
select paternita.figlio, padre, madre  
from    paternita left outer join maternita  
        on paternita.figlio = maternita.figlio
```

NOTA: "outer" è opzionale

```
select paternita.figlio, padre, madre  
from    paternita left join maternita  
        on paternita.figlio = maternita.figlio
```





## Outer join, esempi

```
select paternita.figlio, padre, madre  
from   maternita join paternita  
       on maternita.figlio = paternita.figlio
```

```
select paternita.figlio, padre, madre  
from   maternita left outer join paternita  
       on maternita.figlio = paternita.figlio
```

```
select paternita.figlio, padre, madre  
from   maternita right outer join paternita  
       on maternita.figlio = paternita.figlio
```

```
select nome, padre, madre  
from   persone full outer join maternita on  
       persone.nome = maternita.figlio  
       full outer join paternita on  
       persone.nome = paternita.figlio or  
       maternita.figlio = paternita.figlio
```

## Outer join, esempi

```
select paternita.figlio, padre, madre  
from   maternita join paternita  
       on maternita.figlio = paternita.figlio
```

```
select paternita.figlio, padre, madre  
from   maternita left outer join paternita  
       on maternita.figlio = paternita.figlio
```

```
select paternita.figlio, padre, madre  
from   maternita right outer join paternita  
       on maternita.figlio = paternita.figlio
```

*se il full join fosse solo su  
maternità e paternità perderei  
le persone che non compaiono  
in alcuna delle due relazioni*

```
select nome, padre, madre  
from   persone full outer join maternita on  
       persone.nome = maternita.figlio  
       full outer join paternita on  
       persone.nome = paternita.figlio or  
       maternita.figlio = paternita.figlio
```



# Ordinamento del risultato: order by

Nome e reddito delle persone con meno di trenta anni **in ordine alfabetico**

```
select nome, reddito  
from persone  
where eta < 30  
order by nome
```



ordine  
ascendente

```
select nome, reddito  
from persone  
where eta < 30  
order by nome desc
```



ordine  
discendente



## Ordinamento del risultato: order by

```
select nome, reddito  
from persone  
where eta < 30
```

nome	reddito
Andrea	21
Aldo	15
Filippo	30

```
select nome, reddito  
from persone  
where eta < 30  
order by nome
```


nome	reddito
Aldo	15
Andrea	21
Filippo	30



## Ordinamento del risultato: order by

Nome e reddito delle persone con meno di trenta anni in ordine crescente rispetto a reddito e, a parità di reddito, rispetto a nome

```
select nome, reddito  
from persone  
where eta < 30  
order by reddito, nome
```



ordine crescente  
rispetto a reddito e,  
a parità di reddito,  
rispetto a nome



## Limite alla dimensione del risultato: limit

Si può indicare un limite alla dimensione del risultato (con la clausola **limit** in SQL, con clausole diverse in altri sistemi), al fine di avere come risultato al massimo un prefissato numero di tuple

```
select nome, reddito
from   persone
where  eta < 30
order by nome
limit 2
```



## Limite alla dimensione del risultato: limit

```
select nome, reddito  
from persone  
where eta < 30  
order by nome  
limit 2
```

nome	reddito
Andrea	21
Aldo	15



# Operatori aggregati

Nelle espressioni della target list possiamo avere anche espressioni che calcolano valori a partire da insiemi di ennuple:

– conteggio, minimo, massimo, media, totale

**Sintassi** base (semplificata):

*Funzione ( [ distinct ] EspressioneSuAttributi )*





# Operatori aggregati: count

## Sintassi:

- conta il numero di ennuple:

`count (*)`

- conta i valori di un attributo (considerando i duplicati):

`count (Attributo)`

- conta i valori distinti di un attributo:

`count (distinct Attributo)`



# Operatore aggregato count: esempio e semantica

*Esempio:* Quanti figli ha Franco?

```
select count(*) as NumFigliDiFranco
from   paternita
where  padre = 'Franco'
```

**Semantica:** l'operatore aggregato (**count**), che conta le ennuple, viene applicato al risultato della seguente interrogazione:

```
select *
from   paternita
where  padre = 'Franco'
```



## Risultato di count: esempio

**paternita**

<b>padre</b>	<b>figlio</b>
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo

<b>NumFigliDiFranco</b>
2



## count e valori nulli

```
select count(*)  
from persone
```

Risultato = numero di ennuple  
= 4

```
select count(reddito)  
from persone
```

Risultato = numero di valori  
diversi da NULL  
= 3

```
select count(distinct reddito)  
from persone
```

Risultato = numero di valori  
distinti (escluso  
NULL)  
= 2

**persone**

nome	eta	reddito
Andrea	27	21
Aldo	25	NULL
Maria	55	21
Anna	50	35



## Altri operatori aggregati

### sum, avg, max, min

- ammettono come argomento un attributo o un'espressione (ma non “\*”)
- **sum** e **avg**: argomenti numerici o tempo
- **max** e **min**: argomenti su cui è definito un ordinamento

*Esempio*: media dei redditi dei figli di Franco.

```
select avg(reddito)
from   persone join paternita on
       nome = figlio
where  padre = 'Franco'
```



# Operatori aggregati e valori nulli

```
select avg(reddito) as redditoMedio  
from persone
```

**persone**

nome	eta	reddito
Andrea	27	30
Aldo	25	NULL
Maria	55	36
Anna	50	36

viene  
ignorato

redditoMedio
34



## Operatori aggregati e target list

Un'interrogazione irragionevole (di chi sarebbe il nome?):

```
select nome, max(reddito)
from persone
```

Affinché l'interrogazione sia ragionevole, la **target list** deve essere **omogenea**, ad esempio:

```
select min(eta), avg(reddito)
from persone
```



## Operatori aggregati e raggruppamenti

- Nei casi visti in precedenza, gli operatori aggregati sono applicati all'insieme di tutte le tuple che formano il risultato
- In molti casi, vorremmo che le funzioni di aggregazione venissero applicate a **partizioni delle ennuple** delle relazioni
- Per specificare le partizioni delle ennuple, si utilizza la clausola **group by**:

**group by** *listaAttributi*



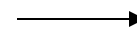
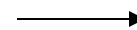
# Operatori aggregati e raggruppamenti

Il numero di figli di ciascun padre

```
select padre, count(*) as NumFigli
from paternita
group by padre
```

**paternita**

padre	figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo



padre	NumFigli
Sergio	1
Luigi	2
Franco	2



# Semantica di interrogazioni con operatori aggregati e raggruppamenti

1. Si esegue l'interrogazione **ignorando la group by** e gli operatori aggregati:

```
select *  
from   paternita
```

2. Si raggruppano le **ennuple che hanno lo stesso valore per gli attributi che compaiono nella group by**, si produce una ennupla del risultato per ogni gruppo, e si applica l'operatore aggregato a ciascun gruppo



## Esercizio 7: group by

Massimo dei redditi per ogni gruppo di persone che sono maggiorenni ed hanno la stessa età (indicando anche l'età)

Esprimere la query in SQL

**persone**

<b>nome</b>	<b>eta</b>	<b>reddito</b>
-------------	------------	----------------



## Esercizio 7: soluzione

Massimo dei redditi per ogni gruppo di persone che sono maggiorenni ed hanno la stessa età (indicando anche l'età)

```
select eta, max(reddito)
from   persone
where  eta > 17
group by eta
```



## Raggruppamenti e target list

In una interrogazione che fa uso di `group by`, dovrebbero comparire solo target list “omogenee”, ovvero target list che comprendono, oltre a funzioni di aggregazione, **solamente** attributi che compaiono nella `group by`.

### Esempio:

- Redditi delle persone, raggruppati per età (**non ragionevole**, perché la target list è disomogenea: potrebbero esistere più valori di reddito per lo stesso gruppo):

```
select eta, reddito
from persone
group by eta
```

- Media dei redditi delle persone, raggruppati per età (**ragionevole**, perché per ogni gruppo c'è una sola media dei redditi):

```
select eta, avg(reddito)
from persone
group by eta
```



## Raggruppamenti e target list

La restrizione di target list omogenea sugli attributi nella `select` vale anche per interrogazioni che semanticamente sarebbero corrette (ovvero, per cui sappiamo che nella base di dati esiste un solo valore dell'attributo per ogni gruppo).

*Esempio:* i padri col loro reddito, e con reddito medio dei figli.

**Target list disomogenea:**

```
select padre, avg(f.reddito), p.reddito
from   persone f join paternita on figlio = nome
       join persone p on padre = p.nome
group by padre
```

**Corretta:**

```
select padre, avg(f.reddito), p.reddito
from   persone f join paternita on figlio = nome
       join persone p on padre = p.nome
group by padre, p.reddito
```



## Target list disomogenea

Abbiamo visto che in una interrogazione che fa uso di **group by**, la target list dovrebbe essere omogenea.

Cosa succede se non lo è? PostgreSQL dà errore, ma alcuni sistemi non segnalano errore e restituiscono uno dei valori che sono associati al valore corrente degli attributi che formano il gruppo.

*Esempio:*

Redditi delle persone, raggruppati per età (**target list disomogenea**, perché potrebbero esistere più valori di reddito per lo stesso gruppo):

```
select eta, reddito
from persone
group by eta
```

ma MySQL, ad esempio, non dà errore, e sceglie per ciascun gruppo uno dei valori di reddito che compare nel gruppo e lo riporta nell'attributo reddito della target list.



## Condizioni sui gruppi

Si possono anche imporre le condizioni di **selezione sui gruppi**. La selezione sui gruppi è **ovviamente diversa** dalla condizione che seleziona le tuple che devono formare i gruppi (clausola **where**). Per effettuare la selezione sui gruppi si usa la clausola **having**, che deve apparire dopo la “**group by**”

*Esempio:* i padri i cui figli hanno un reddito medio maggiore di 25.

```
select padre, avg(f.reddito)
from   persone f join paternita
      on figlio = f.nome
group by padre
having avg(f.reddito) > 25
```





## Esercizio 8: where o having?

I padri i cui figli sotto i 30 anni hanno un reddito medio maggiore di 20



## Esercizio 8: soluzione

I padri i cui figli sotto i 30 anni hanno un reddito medio maggiore di 20

```
select padre, avg(f.reddito)
from   persone f join paternita
      on figlio = f.nome
where  f.eta < 30
group by padre
having avg(f.reddito) > 20
```



## Sintassi, riassumiamo

*SelectSQL ::=*

**select**      *ListaAttributiOEspressioni*  
**from**        *ListaTabelle*  
[ **where**      *CondizioniSemplici* ]  
[ **group by** *ListaAttributiDiRaggruppamento* ]  
[ **having**     *CondizioniAggregate* ]  
[ **order by** *ListaAttributiDiOrdinamento* ]  
[ **limit**      *numero* ]



# Unione, intersezione e differenza

La **select** da sola non permette di fare unioni

Serve un costrutto esplicito:

```
select ...  
union [all]  
select ...
```

Con **union**, i duplicati vengono eliminati (anche in presenza di proiezioni)

Con **union all** vengono mantenuti i duplicati



## Notazione posizionale

```
select padre, figlio
from paternita
union
select madre, figlio
from maternita
```

Quali nomi per gli attributi del risultato? Dipende dal sistema:

- nuovi nomi decisi dal sistema, oppure
- quelli del primo operando, oppure
- ...



## Risultato dell'unione

padre	figlio
Sergio	Franco
Luigi	Olga
Luigi	Filippo
Franco	Andrea
Franco	Aldo
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo



## Notazione posizionale: esempio

```
select padre, figlio  
from paternita  
union  
select madre, figlio  
from maternita
```

```
select padre, figlio  
from paternita  
union  
select figlio, madre  
from maternita
```

queste due interrogazioni sono,  
ovviamente, diverse!



# Ancora sulla notazione posizionale

Con le ridenominazioni non cambia niente:

```
select padre as genitore, figlio
from paternita
union
select figlio, madre as genitore
from maternita
```

La seguente è la query corretta, se vogliamo trattare i padri e le madri come i genitori):

```
select padre as genitore, figlio
from paternita
union
select madre as genitore, figlio
from maternita
```





# Differenza

```
select nome  
from   impiegato  
except  
select cognome as nome  
from   impiegato
```

Nota: **except** elimina i duplicati

Nota: **except all** non elimina i duplicati

Vedremo che la differenza si può esprimere anche con **select** annidate.



# Intersezione

```
select nome
from   impiegato
intersect
select cognome as nome
from   impiegato
```

equivale a

```
select distinct i.nome
from   impiegato i, impiegato j
where  i.nome = j.cognome
```

Nota: **intersect** elimina i duplicati

Nota: **intersect all** non elimina i duplicati



## 3. Il Linguaggio SQL

### 3.1 Definizione dei dati

1. interrogazioni semplici
- 2. definizione dei dati**
3. manipolazione dei dati
4. interrogazioni complesse
5. ulteriori aspetti



# Definizione dei dati in PostgreSQL:

## `create schema`

- La gestione di schemi e databases nei DMBS varia da sistema a sistema.
- PostgreSQL può gestire vari **databases** (un database si crea con **create database**) e prevede l'istruzione **create schema**, che, contrariamente a quanto suggerito dal nome, non serve a dichiarare uno schema propriamente detto (secondo quanto abbiamo detto finora) per tutto il database, ma un cosiddetto **namespace** all'interno di un database.
- Ad un namespace si possono associare relazioni, vincoli, privilegi per gli utenti, ecc. ed operare sugli stessi in modo unitario. Ogni schema ha un nome, e la notazione estesa per gli oggetti in esso definiti è:  
`<nomeschema>.<nomeoggetto>`
- In ogni database si possono quindi definire più schemi, e si possono eseguire operazioni che coinvolgono tabelle di più schemi, a patto di usare il nome esteso della tabella, ovvero  
`<nomeschema>.<nometabella>`
- Al contrario, diversi databases non si “parlano”: PostgreSQL non può eseguire operazioni che coinvolgono tabelle di più databases.

# Definizione dei dati in SQL: `create table`

- L'istruzione più importante del DDL di SQL è  
**`create table`**
  - definisce uno schema di relazione (specificando attributi e vincoli) in uno schema di una base di dati
  - crea un'istanza vuota dello schema di relazione
- Sintassi: **`create table`** *NomeTabella* (  
    *NomeAttributo Dominio [ Vincoli ]*  
    .....  
    *NomeAttributo Dominio [ Vincoli ]*  
    [ *AltriVincoli* ]  
)

## create table: esempio

```
create table Impiegato (  
  Matricola      character(6) primary key,  
  Nome           character(20) not null,  
  Cognome        character(20) not null,  
  Dipart         character(15),  
  Stipendio      numeric(9) default 0,  
  Citta         character(15),  
  foreign key(Dipart) references  
    Dipartimento(NomeDip),  
  unique (Cognome, Nome)  
)
```

nome  
tabella

vincolo

nome  
attributo

dominio  
(o tipo)



# Domini per gli attributi

## • Domini predefiniti

- **Carattere**: singoli caratteri o stringhe, anche di lunghezza variabile
  - **char**(*n*) o **character**(*n*) – stringhe di lunghezza fissa
  - **varchar**(*n*) (o **char varying**(*n*)) – stringhe di lunghezza variabile
  - **nchar**(*n*) e **nvarchar**(*n*) (o **nchar varying**(*n*)) - come sopra ma UNICODE
- **Numerici**: esatti e approssimati
  - **int** o **integer**, **smallint** - interi
  - **numeric**, (o **numeric**(*p*), **numeric**(*p*, *s*)) - valori numerici esatti nonnegativi
  - **decimal**, (o **decimal**(*p*), **decimal**(*p*, *s*)) - valori numerici esatti anche negativi
  - **float**, **float**(*p*), **real**, **double precision** - reali
- **Data, ora, intervalli di tempo**
  - **Date**, **time**, **timestamp**
  - **time with timezone**, **timestamp with timezone**
- **Bit**: singoli bit o stringhe di bit
  - **bit**(*n*)
  - **bit varying**(*n*)
- **Introdotti in SQL:1999**
  - **boolean**
  - **BLOB**, **CLOB**, **NCLOB** (binary/character large object): per grandi immagini e testi



# Definizione dei dati in SQL: create domain

- **Domini definiti dagli utenti**

- L'istruzione

**create domain**

definisce un dominio (semplice) con vincoli e valori di default, utilizzabile in definizioni di relazioni

- Sintassi

```
create domain NomeDominio  
as DominioPreesistente [ Default ] [ Vincoli ]
```

- **Esempio:**

```
create domain Voto  
as smallint default null  
check ( value >=18 and value <= 30 )
```

- **Compatibilità:** il nuovo dominio ed il dominio di partenza (quello che compare dopo la “as”) sono compatibili, ed inoltre i valori del nuovo dominio devono rispettare i vincoli indicati nella definizione





# Vincoli in SQL

Vedremo diversi tipi di vincoli in SQL, sia intrarelazionali, sia interrelazionali.

Ogni vincolo può essere dichiarato con nome esplicito oppure senza nome esplicito (in questo caso il nome viene deciso dal sistema). Per dichiarare un vincolo con il nome occorre fare precedere la sua definizione da

**constraint** <nome del vincolo>

In queste slide ometteremo spesso di assegnare nomi espliciti il nome dei vincoli, per brevità. Ma nella pratica questa possibilità è molto importante: dare un nome ad un vincolo consente di riferirsi ad esso in modo non ambiguo (utile, ad esempio, nella segnalazione che il sistema fa quando viene violato).



## Vincoli intrarelazionali

- **not null** (su singoli attributi)
- **unique**: permette di definire un insieme di attributi come superchiave:
  - singolo attributo:  
**unique** dopo la specifica del dominio
  - più attributi:  
**unique** (*Attributo*, ..., *Attributo*)
- **primary key**: definizione della chiave primaria (una sola chiave primaria, su uno o più attributi); sintassi come per **unique**; implica **not null**
- **check**, per vincoli di tuple o anche più complessi (vedi dopo)

## Vincoli intrarelazionali, esempi

```
create table Impiegato (  
    Matricola      character(6) primary key,  
    Nome           character(20) not null,  
    Cognome        character(20) not null,  
    Dipart         character(15) ,  
    Stipendio      numeric(9) default 0,  
    Citta          character(15) ,  
    foreign key (Dipart) references  
        Dipartimento (NomeDip) ,  
    unique (Cognome, Nome)  
)
```



## primary key, alternative

```
create table Impiegato (  
    Matricola character(6) primary key,  
    ...  
)
```

oppure

```
create table Impiegato (  
    Matricola character(6),  
    ...  
    primary key (Matricola)  
)
```

## Chiavi su più attributi, attenzione

```
create table Impiegato ( ...  
    Nome      character(20) not null,  
    Cognome   character(20) not null,  
    unique (Cognome, Nome)  
)
```

è **diverso** da:

```
create table Impiegato ( ...  
    Nome      character(20) not null unique,  
    Cognome   character(20) not null unique  
)
```



# Vincoli interrelazionali

- **check**, per vincoli complessi
- **references** e **foreign key** permettono di definire vincoli di integrità referenziale

## Sintassi:

- per singoli attributi:

**references** dopo la specifica del dominio

- riferimenti su più attributi:

**foreign key (Attributo, ..., Attributo) references ...**

Gli attributi referenziati nella tabella di arrivo devono formare una chiave (**primary key** o **unique**). Se mancano, il riferimento si intende alla chiave primaria

**Semantica**: ogni combinazione (senza NULL) di valori per gli attributi nella tabella di partenza deve comparire nella tabella di arrivo

- È possibile associare politiche di reazione alla violazione dei vincoli (causate da modifiche sulla tabella esterna, cioè quella cui si fa riferimento)



# Vincoli interrelazionali, esempio

## Infrazioni

<u>Codice</u>	Data	Vigile	Prov	Numero
34321	1/2/95	3987	MI	39548K
53524	4/3/95	3295	TO	E39548
64521	5/4/96	3295	PR	839548
73321	5/2/98	9345	PR	839548

## Vigili

<u>Matricola</u>	Cognome	Nome
3987	Rossi	Luca
3295	Neri	Piero
9345	Neri	Mario
7543	Mori	Gino

# Vincoli interrelazionali, esempio (cont.)

## Infrazioni

<u>Codice</u>	Data	Vigile	Prov	Numero
34321	1/2/95	3987	MI	39548K
53524	4/3/95	3295	TO	E39548
64521	5/4/96	3295	PR	839548
73321	5/2/98	9345	PR	839548

## Auto

<u>Prov</u>	<u>Numero</u>	Cognome	Nome
MI	39548K	Rossi	Mario
TO	E39548	Rossi	Mario
PR	839548	Neri	Luca





## Vincoli interrelazionali, esempio

```
create table Infrazioni (  
    Codice    character(6) not null primary key,  
    Data      date not null,  
    Vigile    integer not null  
                references Vigili(Matricola) ,  
    Provincia character(2) ,  
    Numero    character(6) ,  
    foreign key(Provincia, Numero)  
                references Auto(Provincia,Numero)  
)
```



# Modifiche degli schemi: `alter table`

**alter table**: permette di modificare una tabella

*Esempio:*

```
create table Infrazioni (  
    Codice          character(6) not null primary key,  
    Data            date not null,  
    Vigile           integer not null  
                    references Vigili(Matricola),  
    Provincia        character(2),  
    Numero           character(6)  
)
```

```
alter table Infrazioni  
add constraint MioVincolo foreign key(Provincia, Numero)  
references Auto(Provincia, Numero)
```

È utile per realizzare vincoli di integrità referenziali ciclici: per far sì che R1 referenzi R2 ed R2 referenzi R1 si può definire prima R1 senza vincolo di foreign key (altrimenti si dovrebbe far riferimento ad R2 che non è stata ancora definita), poi R2 con il vincolo di foreign key verso R1, ed infine aggiungere il vincolo di foreign key ad R1 con il comando **alter table**



# Modifiche degli schemi: drop table

**drop table**: elimina una tabella

Sintassi:

```
drop table NomeTabella restrict | cascade
```

*Esempio:*

```
drop table Infrazioni restrict
```

 o semplicemente

```
drop table Infrazioni
```

– elimina la tabella solo se non ci sono riferimenti ad essa

```
drop table Infrazioni cascade
```

 – elimina la tabella e tutte le  
tabella (o più in generale tutti gli oggetti del database) che si  
riferiscono ad essa



# Definizione di indici

## Definizione di indici:

- è rilevante dal punto di vista delle prestazioni
- riguarda il livello fisico, non quello logico
- in passato era importante perché in alcuni sistemi era l'unico mezzo per definire chiavi
- istruzione **create index**
- Sintassi (semplificata):

**create [unique] index NomeIndice on  
NomeTabella Attributo,...,Attributo)**

- *Esempio:*

**create index IndiceIP on  
Infrazioni (Provincia)**



# Catalogo o dizionario dei dati

Ogni sistema relazionale mette a disposizione delle tabelle già definite che raccolgono tutti i dati relativi a:

- **tabelle**
- **attributi**
- ...

Ad esempio, la tabella **Columns** contiene i campi:

- **Column\_Name**
- **Table\_name**
- **Ordinal\_Position**
- **Column\_Default**
- ...



## 3. Il Linguaggio SQL

### 3.2 Manipolazione dei dati

1. interrogazioni semplici
2. definizione dei dati
- 3. manipolazione dei dati**
4. interrogazioni complesse
5. ulteriori aspetti



# Operazioni di aggiornamento in SQL

- operazioni di
  - inserimento: `insert`
  - eliminazione: `delete`
  - modifica: `update`
- di una o più ennuple di una relazione
- sulla base di una condizione che può coinvolgere anche altre relazioni



# Inserimento: sintassi

```
insert into Tabella [ ( Attributi ) ]  
values ( Valori )
```

oppure

*per capirlo dobbiamo  
conoscere l'istruzione  
"select"*

```
insert into Tabella [ ( Attributi ) ]  
select ...
```





# Inserimento: esempio

```
insert into persone values ('Mario',25,52)
```

```
insert into persone(nome, eta, reddito)  
values ('Pino',25,52)
```

```
insert into persone(nome, reddito)  
values ('Lino',55)
```

*per capirlo dobbiamo  
conoscere l'istruzione  
“select”*

```
insert into persone (nome)  
select padre  
from paternita  
where padre not in (select nome from persone)
```



## Inserimento: commenti

- l'ordinamento degli attributi (se presente) e dei valori è significativo
- le due liste di attributi e di valori debbono avere lo stesso numero di elementi
- se la lista di attributi è omessa, si fa riferimento a tutti gli attributi della relazione, secondo l'ordine con cui sono stati definiti nella «create table»
- se la lista di attributi non contiene tutti gli attributi della relazione, per gli altri viene inserito il valore di default o il valore nullo (che deve essere permesso)



# Eliminazione di ennuple

Sintassi:

```
delete from Tabella [ where Condizione ]
```

*Esempi:*

```
delete from persone  
where eta < 35
```

```
delete from paternita  
where figlio not in
```

```
(select nome from persone)
```

*per capirlo dobbiamo  
conoscere l'istruzione  
"select"*



## Eliminazione: commenti

- elimina le ennuple che soddisfano la condizione
- può causare (se i vincoli di integrità referenziale sono definiti con politiche di reazione **cascade**) eliminazioni da altre relazioni
- ricordare: se la **where** viene omessa, si intende **where true**

# Modifica di ennuple

- **Sintassi:**

**update** *NomeTabella*

**set** *Attributo* = < *Espressione* | **select** ... | **null** | **default** >  
[ **where** *Condizione* ]

- **Semantica:** vengono modificate le ennuple della tabella che soddisfano la condizione “where”

- *Esempi:*

```
update persone set reddito = 45  
where nome = 'Piero'
```

```
update persone set reddito = reddito * 1.1  
where eta < 30
```



## 3. Il Linguaggio SQL

### 3.2 Manipolazione dei dati

1. interrogazioni semplici
2. definizione dei dati
3. manipolazione dei dati
- 4. interrogazioni complesse**
5. ulteriori aspetti



## Interrogazioni annidate

- Nelle condizioni atomiche può comparire una **select** (sintatticamente, deve comparire tra parentesi).
- In particolare, le condizioni atomiche permettono:
  - il confronto fra un attributo (o più attributi) e il risultato di una sottointerrogazione
  - quantificazioni esistenziali



## Interrogazioni annidate: esempio

Nome e reddito del padre di Franco.

```
select  nome, reddito
from    persone, paternita
where   nome = padre and figlio = 'Franco'
```

```
select  nome, reddito
from    persone
where   nome = (select padre
                  from    paternita
                  where   figlio = 'Franco')
```



## Interrogazioni annidate: operatori

Il risultato di una interrogazione annidata può essere messo in relazione nella clausola **where** mediante diversi **operatori**:

- uguaglianza o altri operatori di confronto (il risultato della interrogazione annidata deve essere unico)
- se non si è sicuri che il risultato sia unico, si può far precedere l'interrogazione annidata da:
  - **any**: vero, se il confronto è vero per **una qualunque** delle tuple risultato dell'interrogazione annidata
  - **all**: vero, se il confronto è vero per **tutte** le tuple risultato dell'interrogazione annidata
- l'operatore **in**, che è equivalente a **=any**
- l'operatore **not in**, che è equivalente a **<>all**
- l'operatore **exists**



## Interrogazioni annidate: esempio

Nome e reddito dei padri di persone che guadagnano più di 20 milioni.

```
select distinct p.nome, p.reddito
from   persone p, paternita, persone f
where  p.nome = padre and figlio = f.nome
       and f.reddito > 20
```

```
select nome, reddito
from   persone
where  nome = any
```

```
(select padre
   from   paternita, persone
  where  figlio = nome
        and reddito > 20)
```

padri di persone  
che guadagnano  
più di 20 milioni



# Interrogazioni annidate: esempio

Nome e reddito dei padri di persone che guadagnano più di 20 milioni.

```
select nome, reddito
from persone
where nome in (select padre
               from persone
               where reddito > 20)
```

padri di persone  
che guadagnano più di  
20 milioni

```
select nome, reddito
from persone
where nome in (select padre
               from paternita
               where figlio in (select nome
                               from persone
                               where reddito > 20)
               )
```

persone che  
guadagnano più  
di 20 milioni



## Interrogazioni annidate: esempio di all

Persone che hanno un reddito maggiore del reddito di tutte le persone con meno di 30 anni.

```
select nome
from   persone
where  reddito > all ( select reddito
                        from persone
                        where eta < 30 )
```



# Interrogazioni annidate: esempio di **exists**

L'operatore **exists** forma una espressione che è vera se il risultato della sottointerrogazione **non è vuota**.

*Esempio*: le persone che hanno almeno un figlio.

```
select *  
from   persone p  
where  exists (select *  
                from   paternita  
                where  padre = p.nome)  
       or  
       exists (select *  
                from   maternita  
                where  madre = p.nome)
```

Si noti che l'attributo **nome** si riferisce alla relazione nella clausola **from**.



## Esercizio 9: interrogazioni annidate

Nome ed età delle madri che hanno almeno un figlio minorenni.

**Soluzione 1:** un join per selezionare nome ed età delle madri, ed una sottointerrogazione per la condizione sui figli minorenni.

**Soluzione 2:** due sottointerrogazioni e nessun join.



## Esercizio 9: soluzione 1

Nome ed età delle madri che hanno almeno un figlio minorenni.

```
select nome, eta
from   persone, maternita
where  nome = madre and
       figlio in (select nome
                  from   persone
                  where  eta < 18)
```



## Esercizio 9: soluzione 2

Nome ed età delle madri che hanno almeno un figlio minorenni.

```
select nome, eta
from persone
where nome in (select madre
                from maternita
                where figlio in (select nome
                                from persone
                                where eta<18))
```





## Interrogazioni annidate, commenti

- La forma annidata può porre problemi di efficienza (i DBMS non sono bravissimi nella loro ottimizzazione), ma talvolta è più leggibile.
- In alcuni sistemi le sottointerrogazioni non possono contenere operatori insiemistici (“ovvero, in tali sistemi l’unione si fa solo al livello esterno”), ma la limitazione non è significativa.



# Interrogazioni annidate: semantica e visibilità

- **Semantica:** l'interrogazione interna viene eseguita una volta **per ciascuna ennupla** dell'interrogazione esterna
- Vale la classica regola di **visibilità** dei linguaggi di programmazione:
  - Una variabile  $X$  è visibile in una “select”  $B$  se  $X$  è definita in  $B$  oppure se  $X$  è (ricorsivamente) visibile nella “select” in cui  $B$  è definita, a meno che  $X$  non sia mascherata da una variabile in  $B$  con lo stesso nome di  $X$ .
  - In altre parole, si può fare riferimento a variabili definite nello stesso blocco o in blocchi più esterni, a meno che esse non siano mascherate da definizioni di variabili di uguale nome. Ovviamente, se un nome di variabile (o tabella) è omesso, si assume riferimento alla variabile (o tabella) più “vicina”



## Interrogazioni annidate: visibilità

Le persone che hanno almeno un figlio.

```
select *  
from persone  
where exists (select *  
              from paternita  
              where padre = nome)  
or  
exists (select *  
        from maternita  
        where madre = nome)
```

L'attributo **nome** si riferisce alla relazione **persone** nella clausola **from**.



## Ancora sulla visibilità

Attenzione alle regole di visibilità; questa interrogazione è **scorretta**:

```
select *  
from impiegato  
where dipart in (select nome  
                  from dipartimento D1  
                  where nome = 'Produzione')  
  
or  
dipart in (select nome  
            from dipartimento D2  
            where D2.citta = D1.citta)
```

**impiegato**

nome	cognome	dipart
------	---------	--------

**dipartimento**

nome	indirizzo	citta
------	-----------	-------



## Visibilità: variabili in blocchi interni

Nome e reddito dei padri di persone che guadagnano più di 20 milioni,  
**con indicazione del reddito del figlio.**

```
select distinct p.nome, p.reddito, f.reddito
from   persone p, paternita, persone f
where  p.nome = padre and figlio = f.nome
       and f.reddito > 20
```

In questo caso l'interrogazione annidata "intuitiva" **non è corretta:**

```
select nome, reddito, f.reddito
from persone
where nome in (select padre
               from paternita
               where figlio in (select nome
                               from persone f
                               where f.reddito > 20))
```



# Interrogazioni annidate e correlate

Può essere necessario usare in blocchi interni variabili definite in blocchi esterni; si parla in questo caso di interrogazioni annidate e **correlate**.

*Esempio:* i padri i cui figli guadagnano tutti più di venti milioni.

```
select distinct padre
from paternita z
where not exists
    (select *
     from paternita w join persone on w.figlio = nome
     where w.padre = z.padre and reddito <= 20)
```



## **Esercizio 10: interrogazioni annidate e correlate**

Nome ed età delle madri che hanno almeno un figlio la cui età differisce meno di 20 anni dalla loro.



## Esercizio 10: soluzione

Nome ed età delle madri che hanno almeno un figlio la cui età differisce meno di 20 anni dalla loro.

```
select nome, eta
from   persone p, maternita
where  nome = madre and
       figlio in (select nome
                  from   persone
                  where  p.eta - eta < 20)
```





## Differenza mediante annidamento

```
select nome from impiegato
```

```
except
```

```
select cognome as nome from impiegato
```

```
select nome
```

```
from impiegato
```

```
where nome not in (select cognome  
                    from impiegato)
```



## Intersezione mediante annidamento

```
select nome from impiegato
intersection
select cognome from impiegato
```

```
select nome
from   impiegato
where  nome in (select cognome
                from impiegato)
```



## Esercizio 11: annidamento e funzioni

La persona (o le persone) con il reddito massimo.



## Esercizio 11: soluzione

La persona (o le persone) con il reddito massimo.

```
select *  
from persone  
where reddito = (select max(reddito)  
                 from persone)
```

Oppure:

```
select *  
from persone  
where reddito >= all (select reddito  
                     from persone)
```



## Interrogazioni annidate: condizione su più attributi

Le persone che hanno la coppia (età, reddito) diversa da tutte le altre persone.

```
select *  
from persone p  
where (eta,reddito) not in  
      (select eta, reddito  
       from persone  
       where nome <> p.nome)
```



# Interrogazioni annidate nella clausola from

Finora abbiamo parlato di query annidate nella clausola where. Ma anche nella clausola from possono apparire query racchiuse tra parentesi, come ad esempio

```
select p.padre  
from paternita p, (select nome  
                   from persone  
                   where eta > 30) f  
where f.nome = p.figlio
```

“query derivata”  
o “vista inline”

Una **vista** è una tabella **le cui tuple sono derivate da altre tabelle mediante una interrogazione.**

La **semantica** è ovvia: la tabella il cui alias è **f**, definita come query annidata nella clausola **from**, invece che essere una tabella della base di dati, è una vista calcolata mediante la associata query **select** racchiusa tra parentesi.



## Importanza delle “viste inline”

Supponiamo di avere le seguenti relazioni

EsamiTriennale(matricola, corso, voto)

EsamiMagistrale(matricola, corso, voto)

e di volere la media dei voti di tutti gli esami (sia nella triennale sia nella magistrale) degli studenti.

```
select avg(v.voto)
from (select matricola,voto
      from EsamiTriennale
      union
      select matricola, voto
      from EsamiMagistrale) v
group by v.matricola
```



## Altro modo di definire e usare le “viste inline”

Supponiamo di avere le seguenti relazioni

EsamiTriennale(matricola, corso, voto)

EsamiMagistrale(matricola, corso, voto)

e di volere la media dei voti di tutti gli esami (sia nella triennale sia nella magistrale) degli studenti.

**with** miavista **as**

```
(select matricola, voto  
from EsamiTriennale  
union  
select matricola, voto  
from EsamiMagistrale)
```

```
select avg(miavista.voto)  
from miavista  
group by miavista.matricola
```



## Interrogazioni annidate nella target list

Finora abbiamo parlato di query annidate nella clausola where o nella clausola from. Ma anche **nella target list possono apparire query racchiuse tra parentesi**, come ad esempio

```
select p.nome, (select count(*) from persone where eta=p.eta)
from persone p
where reddito > 10
```

“vista inline” nella  
target list

La query riportata qui sopra calcola il nome di ogni persona p con reddito maggiore di 10 ed accanto a tale nome mostra anche il numero di persone che hanno la stessa età della persona p

La **semantica** è ovvia: per ogni tupla selezionata della tabella calcolata nella clausola from e selezionata dalla clausola where, viene calcolata la target list, e questo richiede l'esecuzione di tutte le vista inline che troviamo nella target list.



## 3. Il Linguaggio SQL

### 3.4 Ulteriori aspetti

1. interrogazioni semplici
2. definizione dei dati
3. manipolazione dei dati
4. interrogazioni complesse
- 5. ulteriori aspetti**



## Vincoli di integrità generici: check

Per specificare vincoli di ennumera o vincoli più complessi su una sola tabella:

**check** (*Condizione*)

```
create table impiegato
( matricola character(6) ,
  cognome character(20) ,
  nome character(20) ,
  sesso character not null check (sesso in ('M' , 'F' ))
  stipendio integer,
  superiore character(6) ,
  check (stipendio <= (select stipendio
                        from   impiegato j
                        where  superiore = j.matricola))
)
```

Purtroppo, gli attuali DBMS (compreso PostgreSQL) accettano “check” nella “create table” solo se esse non contengono query annidate. Ne segue che la seconda “check” dell’esempio non viene accettata da PostgreSQL. Altri DBMS, come MySQL, accettano la clausola “check”, ma la ignorano!



## Vincoli di integrità generici: asserzioni

Specifica vincoli a livello di schema. Sintassi:

```
create assertion NomeAss check ( Condizione )
```

*Esempio:*

```
create assertion AlmenoUnImpiegato  
check (1 <= (select count(*)  
             from impiegato))
```

Purtroppo, gli attuali DBMS (compreso PostgreSQL) non accettano istruzioni di tipo “create assertion”.



## Viste

- Come abbiamo già detto, una vista è una tabella **la cui istanza è derivata da altre tabelle mediante una interrogazione.**

```
create view NomeVista [(ListaAttributi)] as SelectSQL
```

- Le viste sono tabelle virtuali: solo quando vengono utilizzate (ad esempio in altre interrogazioni) la loro istanza viene calcolata.

- *Esempio:*

```
create view ImpAmmin(Mat,Nome,Cognome,Stip)  
as  
select Matricola, Nome, Cognome, Stipendio  
from    Impiegato  
where   Dipart = 'Amministrazione' and  
        Stipendio > 10
```



# Un'interrogazione con annidamento nella having

- Voglio sapere l'età delle persone cui corrisponde il massimo reddito (come somma dei redditi delle persone che hanno quella età).
- Assumendo che non ci siano valori nulli in reddito, usando l'annidamento nella having, otteniamo questa soluzione:

```
select eta
from   persone
group by eta
having sum(reddito) >= all (select sum(reddito)
                           from persone
                           group by eta)
```

- Un'altro metodo è definire una vista.



## Soluzione con le viste

```
create view etaReddito(eta,totaleReddito) as  
  select eta, sum(reddito)  
  from   persone  
  group by eta
```

```
select eta  
from   etaReddito  
where  totaleReddito = (select max(totaleReddito)  
                        from etaReddito)
```



## Tabelle e viste temporanee

- In molti DBMS basati su SQL è possibile specificare che una tabella o una vista che stiamo creando è temporanea, ovvero che sparirà alla fine della sessione di connessione con il sistema
- Ad esempio: `create temporary table` MiaTabella ... oppure `create temporary view` MiaVista ...
- Possiamo anche cancellare con la `drop table` o `drop view` la tabella o la vista così creata prima della fine della sessione, ma in ogni caso essa sarà eliminata alla fine della sessione, se non l'abbiamo fatto prima.
- Le tabelle o viste temporanee possono essere utili per salvare nella base di dati i risultati di una query in modo temporaneo.





# Privilegi

- Un privilegio è caratterizzato da:
  - la risorsa cui si riferisce
  - l'utente che concede il privilegio
  - l'utente che riceve il privilegio
  - l'azione che viene permessa
  - la trasmissibilità del privilegio
- Tipi di privilegi
  - **insert**: permette di inserire nuovi oggetti (ennuple)
  - **update**: permette di modificare il contenuto
  - **delete**: permette di eliminare oggetti
  - **select**: permette di leggere la risorsa
  - **references**: permette la definizione di vincoli di integrità referenziale verso la risorsa (può limitare la possibilità di modificare la risorsa)
  - **usage**: permette l'utilizzo in una definizione (per esempio, di un dominio)



## grant e revoke

- **Concessione** di privilegi:

`grant < Privileges | all privileges > on  
Resource to Users [ with grantOption ]`

- *grantOption* specifica se il privilegio può essere trasmesso ad altri utenti

`grant select on Dipartimento to Giuseppe`

- **Revoca** di privilegi:

`revoke Privileges on Resource from Users  
[ restrict | cascade ]`



# Transazione

- Insieme di operazioni da considerare indivisibile (“atomico”), corretto anche in presenza di concorrenza, e con effetti definitivi.
- Proprietà (“**ACIDe**”):
  - **A**tomicità
  - **C**onsistenza
  - **I**solamento
  - **D**urabilità (persistenza)



## Le transazioni sono ... atomiche

- La sequenza di operazioni sulla base di dati viene eseguita per intero o per niente.

*Esempio:* trasferimento di fondi da un conto A ad un conto B:  
o si fa sia il prelevamento da A sia il versamento su B, o  
nessuno dei due.



## Le transazioni sono ... consistenti

- Al termine dell'esecuzione di una transazione, i vincoli di integrità debbono essere soddisfatti.
- “Durante” l'esecuzione si può chiedere di accettare violazioni di vincoli (si veda più avanti il comando SET CONSTRAINTS DEFERRED), almeno per quei vincoli definiti “deferrable”
- Se anche una violazione rimane alla fine, allora la transazione deve essere annullata per intero (“abortita”) .



## Le transazioni sono ... isolate

- L'effetto di transazioni concorrenti deve essere coerente (ad esempio “equivalente” all'esecuzione separata).

*Esempio:* se due assegni emessi sullo stesso conto corrente vengono incassati contemporaneamente si deve evitare di trascurarne uno.



# I risultati delle transazioni sono durevoli

- La conclusione positiva di una transazione corrisponde ad un impegno (in inglese **commit**) a mantenere traccia del risultato in modo definitivo, anche in presenza di guasti e di esecuzione concorrente.



# Transazioni in SQL

## Istruzioni fondamentali

- **begin** (o **begin transaction**): specifica l'inizio della transazione (le operazioni non vengono eseguite sulla base di dati)
- **commit** (o **commit work**, o **end**, o **end transaction**): le operazioni specificate a partire dal **begin** vengono rese permanenti sulla base di dati
- **rollback** (o **rollback work**): si disfano gli effetti delle operazioni specificate dall'ultimo **begin**





## Esempio di transazione in SQL

```
begin;  
    update ContoCorrente  
        set Saldo = Saldo - 10  
        where NumeroConto = 12345;  
    update ContoCorrente  
        set Saldo = Saldo + 10  
        where NumeroConto = 55555;  
commit;
```



# Vincoli di foreign key: reazioni ad aggiornamenti

Specifica di vincolo di foreign key nella tabella T1 verso la tabella T2:

```
FOREIGN KEY [Nome] (Attributi) REFERENCES Tabella [(Attributo)]  
[ON DELETE (NO ACTION | CASCADE | RESTRICT | SET DEFAULT | SET NULL) ]  
[ON UPDATE (NO ACTION | CASCADE | RESTRICT | SET DEFAULT | SET NULL) ]
```

- **no action**: se si tenta di cancellare/aggiornare la tupla “padre” (quella nella tabella referenziata T2) di una tupla della tabella T1, si genera un errore al momento della verifica del vincolo. Si noti che, se il vincolo è deferred, questo momento è la fine della transazione. Si noti che **no action** è il default: cioè è l’opzione che vale se non si specifica nulla.
- **restrict**: se si tenta di cancellare/aggiornare la tupla “padre”, si genera un errore immediato (se il comando è all’interno della transazione, non si aspetta il momento la fine della transazione, nemmeno se il vincolo è “deferred”)
- **cascade**: quando si cancella/aggiorna la tupla “padre”, si cancella anche la tupla della tabella T1 che referencia la tupla “padre”
- **set default**: quando si cancella/aggiorna la tupla “padre”, si memorizza il valore di default nell’attributo di T1 che è definito come foreign key
- **set null**: quando si cancella/aggiorna la tupla “padre”, si memorizza il valore NULL nell’attributo di T1 che è definito come foreign key



## Transazioni in SQL: vincoli “deferred”

- Un vincolo “deferrable” è un vincolo che si può definire “deferred” (opposto di IMMEDIATE) all’interno di una transazione. Quando un vincolo è definito deferred in una transazione, esso viene controllato alla fine della transazione, invece che immediatamente.
- Per specificare un vincolo “deferrable” si deve aggiungere alla definizione di vincolo la parola DEFERRABLE (aggiungendo, se vogliamo, anche INITIALLY DEFERRED oppure INITIALLY IMMEDIATE – considerando che il default è INITIALLY IMMEDIATE). Possiamo anche usare NOT DEFERRABLE, che è il default.
- Se un vincolo di nome <nome> è definito come “deferrable”, allora si può dare il comando all’interno di una transazione:

SET CONSTRAINTS <nome> DEFERRED

ed il vincolo di nome <nome> verrà controllato solo alla fine della transazione. Al posto di <nome> si può specificare ALL, se vogliamo che tutti i vincoli DEFERRABLE siano deferred.

- Se vogliamo tornare alla situazione IMMEDIATE, possiamo dare il comando SET CONSTRAINTS <nome> IMMEDIATE, ma attenzione: quando il sistema esegue questa istruzione controlla i vincoli specificati, senza aspettare la fine della transazione.



## Esempio

Definiamo due relazioni con vincoli di foreign key definiti mutuamente:

```
create table persona(cf varchar(100) primary key, cittanascita varchar(30));
```

```
create table citta(nome varchar(30) primary key, sindaco varchar(100)  
                 constraint vincolo1 references persona(cf));
```

```
alter table persona add constraint vincolo2 foreign key(cittanascita)  
                 references citta(nome);
```

Si noti che né cittanascita né sindaco è NOT NULL. Inseriamo ora due città ed una persona. Possiamo usare NULL per non violare i vincoli:

```
insert into citta values('Roma',null), ('Milano',null);
```

```
insert into persona values('100','Roma');
```



## Esempio

Ma attenzione: se eseguiamo la cancellazione della città di nome 'Roma', otteniamo un errore:

`delete from citta where nome = 'Roma';`

*ERROR: update or delete on table "citta" violates foreign key constraint "vincolo2" on table "persona"*

*DETAIL: Key (nome)=(Roma) is still referenced from table "persona".*

Come risolviamo? Ci sono almeno tre metodi:

- 1) usiamo NULL per evitare che rimangano tuple che referenziano Roma
- 2) cambiamo la create table citta definendo il vincolo di foreign key cascade e cancelliamo automaticamente le persone nate a Roma
- 3) cambiamo la create table definendo il vincolo di foreign key verso «persona» come «deferrable» ed usiamo una transazione dentro la quale dichiariamo il vincolo «deferred»



## Esempio: metodo 1)

Per eseguire la cancellazione della città Roma:

```
update persona set cittanascita = null where CF = '100';
```

```
delete from citta where nome = 'Roma';
```

Ovviamente questa soluzione presuppone che il valore null sia accettabile nel campo cittanascita



## Esempio: metodo 2)

Sostituiamo a quelle di prima le seguenti definizioni delle create table, che ora usano «on delete cascade» (in particolare, in questo esempio è fondamentale che «on delete cascade» sia associato al vincolo di foreign key che appare nella relazione città):

```
create table persona(cf varchar(100) primary key, cittanascita varchar(30));
create table città(nome varchar(30) primary key, sindaco varchar(100)
                  constraint vincolo1 references persona(cf) on delete cascade;
alter table persona add constraint vincolo2 foreign key(cittanascita)
                  references città(nome) on delete cascade;
insert into città values('Roma',null), ('Milano',null);
insert into persona values('100','Roma');
```

Ora cancelliamo la città di Roma, con una semplice «delete»:

```
delete from città where nome = 'Roma';
```

e non otteniamo un errore, ma otteniamo la cancellazione delle persone che avevano la città di nascita pari a 'Roma'.



## Esempio: metodo 3)

Sostituiamo a quelle di prima le seguenti definizioni delle create table, che ora dichiarano i vincoli di foreign key «deferrable»:

```
create table persona(cf varchar(100) primary key, cittanascita varchar(30));  
create table citta(nome varchar(30) primary key, sindaco varchar(100)  
    constraint vincolo1 references persona(cf) deferrable);  
alter table persona add constraint vincolo2 foreign key(cittanascita)  
    references citta(nome) deferrable;  
insert into citta values('Roma',null), ('Milano',null);  
insert into persona values('100','Roma');
```

Per cancellare la città di Roma, usiamo una transazione in cui definiamo vincolo2 come deferred (che quindi verrà controllato alla fine della transazione):

```
begin;  
SET CONSTRAINTS vincolo2 DEFERRED;  
delete from citta where nome = 'Roma';  
update persona set cittanascita = null where CF = '100';  
end;
```