



Basi di dati

Maurizio Lenzerini

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

Anno Accademico 2024/2025

<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 (diverse) 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** (ossia “dati sui dati”, il catalogo – vedi dopo)

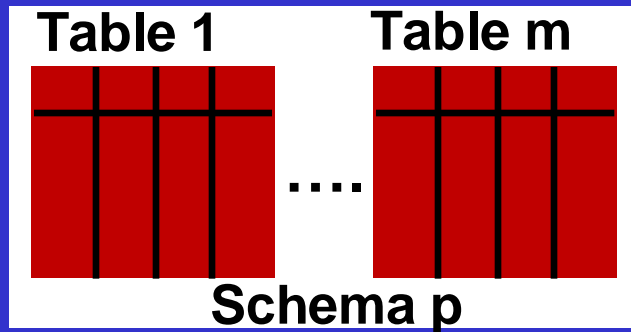
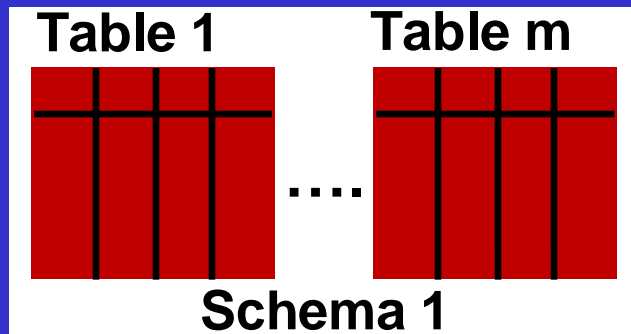
SQL e modello relazionale

- **Attenzione:** una tabella in SQL è definita come un multiinsieme di tuple
- In particolare, se una tabella non ha una primary key o un insieme di attributi definiti come unique (vedi dopo), allora potranno comparire due tuple uguali nella tabella; ne segue che una tabella SQL **non** è in generale una relazione
- Se invece una tabella ha una primary key o comunque un insieme di attributi definiti come superchiavi, allora non potranno mai comparire nella tabella due tuple uguali e quindi in questo caso la tabella è una relazione. Per questo, è consigliabile definire almeno una primary key per ogni tabella: per poi trattare quella tabella coerentemente con la definizione del modello relazionale
- Si noti comunque che, anche partendo da tabelle senza duplicati, eseguendo delle query potremo ottenere tabelle che i duplicati li hanno.

Basi di dati, schemi e tabelle in PostgreSQL

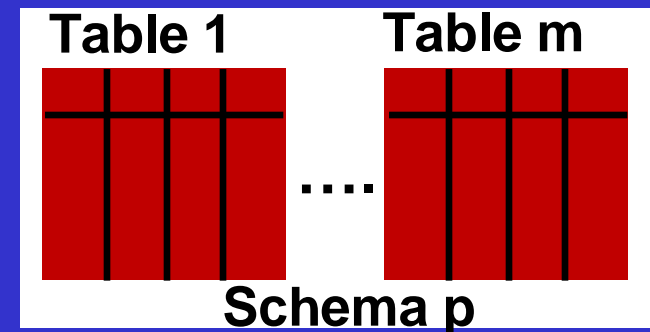
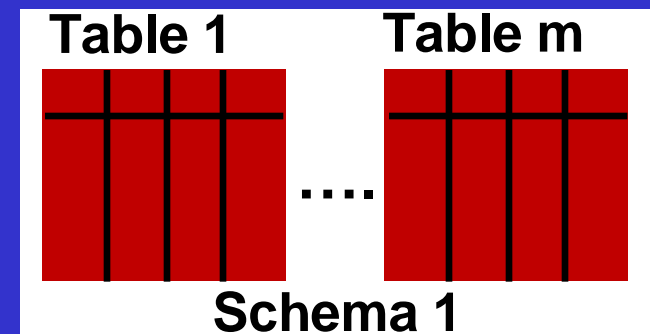
Installazione server PostgreSQL

Database 1



.....

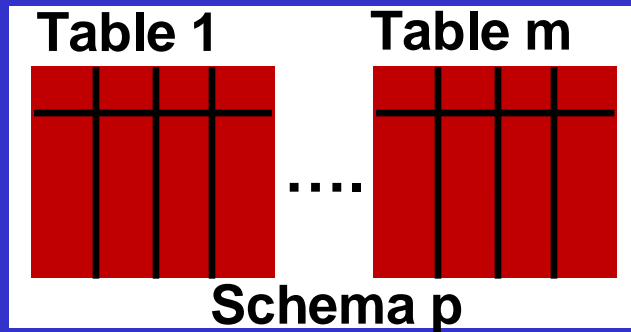
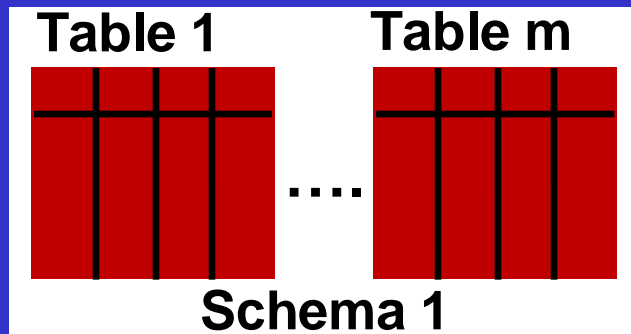
Database n



Basi di dati, schemi e tabelle in PostgreSQL

Installazione server PostgreSQL

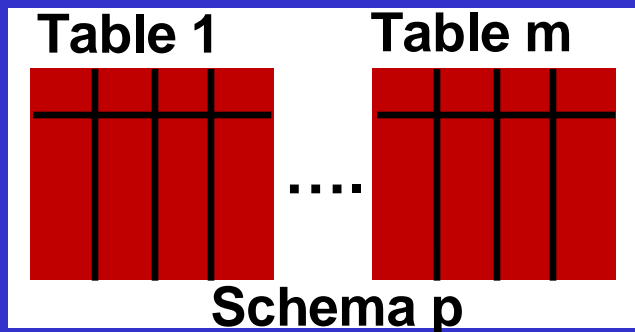
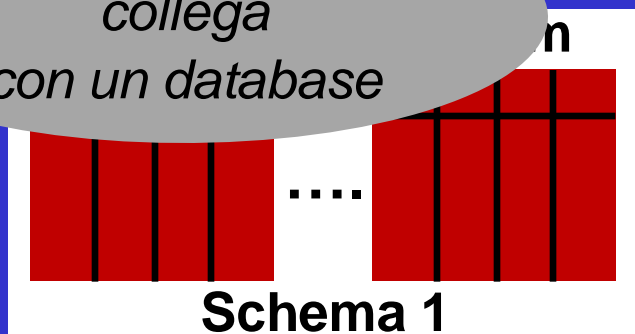
Database 1



.....

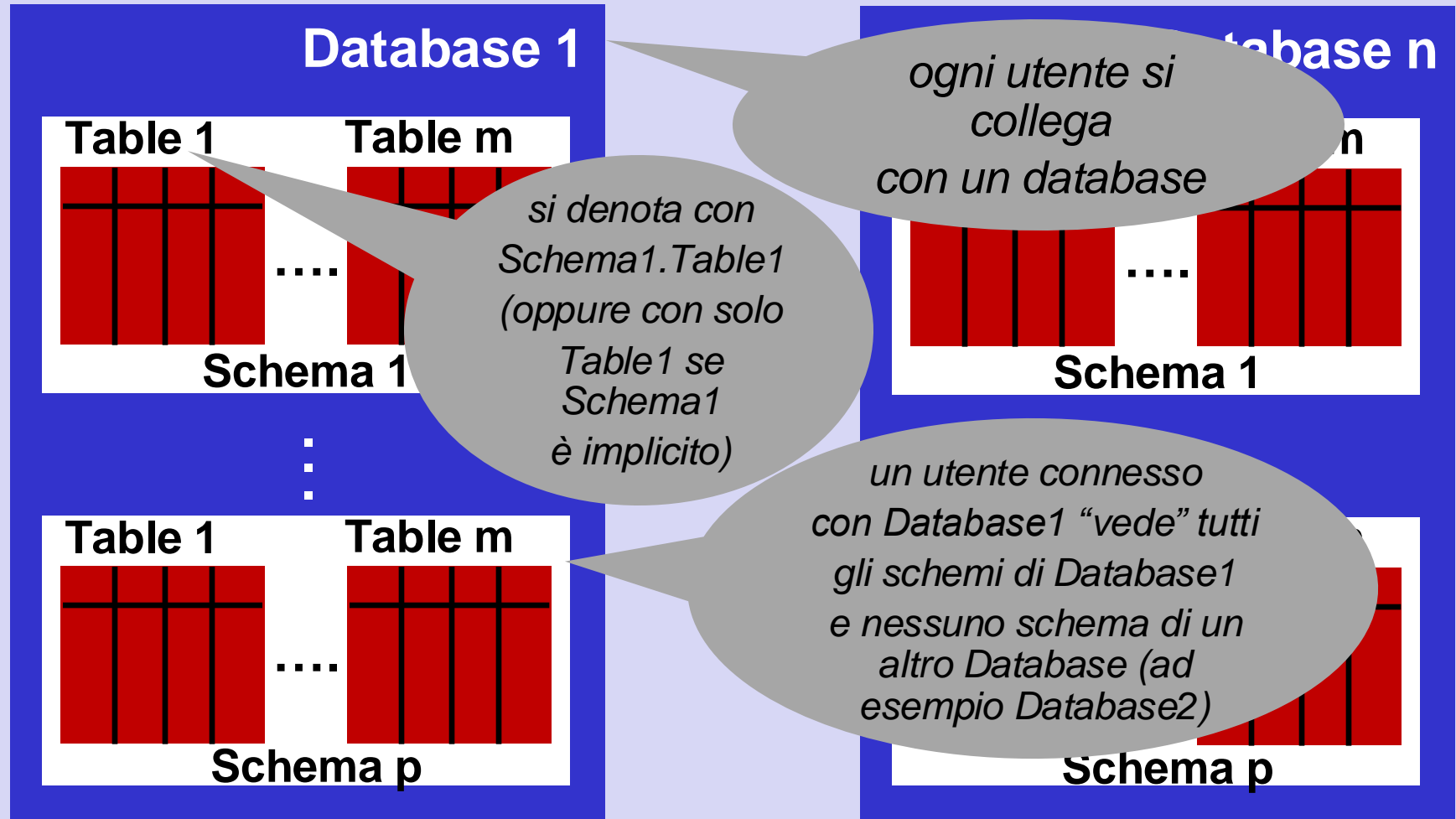
Database n

*ogni utente si
collega
con un database*



Basi di dati, schemi e tabelle in PostgreSQL

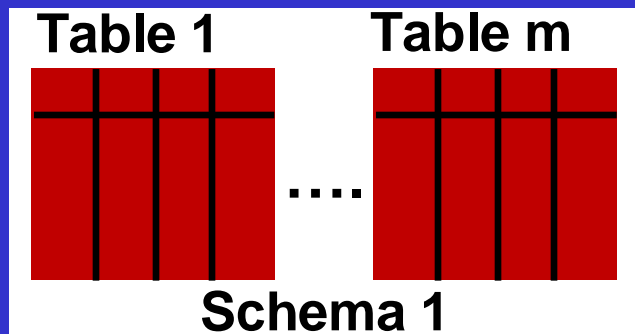
Installazione server PostgreSQL



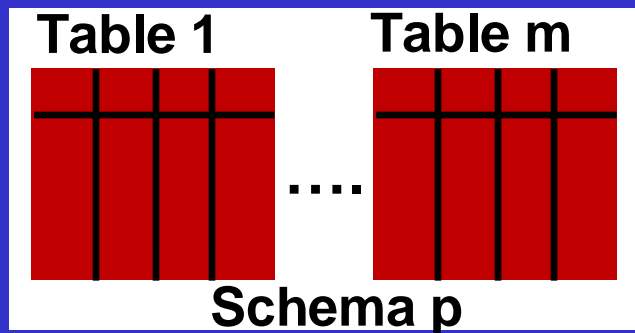
Basi di dati, schemi e tabelle in PostgreSQL

Installazione server PostgreSQL

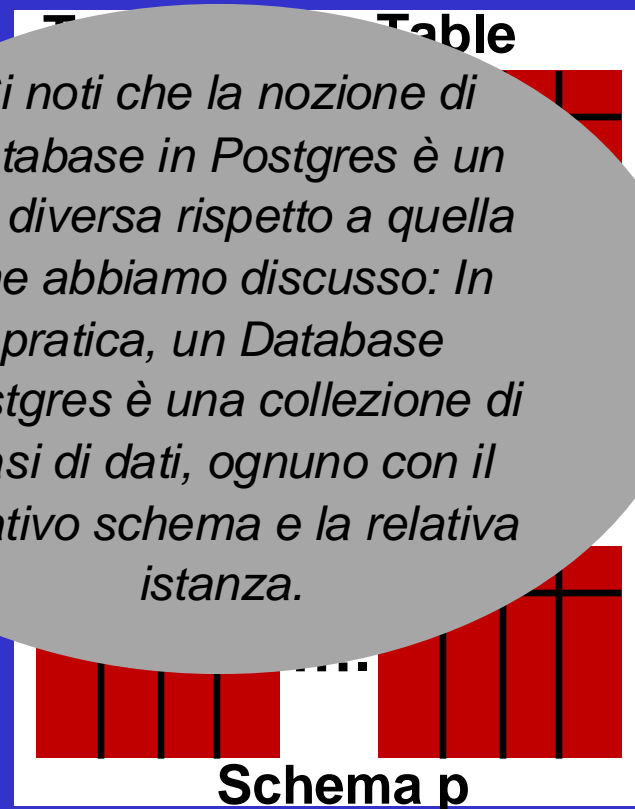
Database 1



⋮



Database n



Si noti che la nozione di Database in Postgres è un po' diversa rispetto a quella che abbiamo discusso: In pratica, un Database Postgres è una collezione di basi di dati, ognuno con il relativo schema e la relativa istanza.

Osservazione importante

Nel seguito di queste slides illustreremo **gli aspetti essenziali di SQL**, puntando a capire i concetti che lo caratterizzano e **NON tutti i dettagli**.

Nel momento in cui un utente del linguaggio lo utilizza in progetti reali deve necessariamente acquisire maggiori dettagli e giungere ad una più approfondita conoscenza del linguaggio, in tutti i suoi aspetti. Ma questo sarà possibile solo se avrà acquisito e compreso i concetti essenziali che qui miriamo ad impartire.

Come si fa ad acquisire la conoscenza su tutti i dettagli? Si deve consultare il manuale del linguaggio, facendo in particolare riferimento alla formulazione del linguaggio nel DBMS che viene utilizzato. Tali manuali si trovano gratuitamente in rete.

Si invitano gli studenti ad abituarsi a consultare il manuale di SQL, facendo riferimento ad esempio alla sua realizzazione in PostgreSQL. Anche imparare a consultare i manuali fa parte della preparazione di un buon ingegnere informatico.



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

Convenzioni sui nomi

- Ogni **tabella** si denota con *NomeSchema . NomeTabella*
- Quando l'ambiguità sullo schema non sussiste (per esempio quando istruiamo il sistema a fare riferimento una volta per tutte ad uno specifico schema, che diventa implicito), si può omettere *NomeSchema .* e scrivere semplicemente *NomeTabella*
- Ogni **attributo** di una tabella si denota con *NomeSchema . NomeTabella . Attributo*
- Quando l'ambiguità sullo schema non sussiste si può ancora una volta omettere *NomeSchema .* e scrivere *NomeTabella . Attributo*
- Quando anche l'ambiguità sulla tabella non sussiste (ad esempio all'interno di una query in cui si usa una sola tabella con quel nome), si può omettere anche *NomeTabella .* e scrivere semplicemente *Attributo*

Istruzione `select` (versione elementare)

- L'istruzione di interrogazione in SQL è

`select`

che definisce una interrogazione (query) e restituisce il risultato della valutazione di quella query sulla base di dati in forma di tabella. La sua forma elementare è:

<code>select</code>	<i>Attributo,...,Attributo</i>
<code>from</code>	<i>Tabella</i>
<code>where</code>	<i>Condizione</i>

Istruzione `select` (versione elementare)

- L'istruzione di interrogazione in SQL è

`select`

che definisce una interrogazione (query) e **restituisce il risultato della valutazione di quella query sulla base di dati in forma di tabella**. La sua forma elementare è:

```
select Attributo,...,Attributo  
from    Tabella  
where   Condizione
```



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

Istruzione `select` (versione elementare)

- L'istruzione di interrogazione in SQL è

`select`

che definisce una interrogazione (query) e **restituisce il risultato della valutazione di quella query sulla base di dati in forma di tabella**. La sua forma elementare è:

```
select  Attributo,...,Attributo  
from    Tabella  
where   Condizione
```



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

Istruzione `select` (versione elementare)

- L'istruzione di interrogazione in SQL è

`select`

che definisce una interrogazione (query) e **restituisce il risultato della valutazione di quella query sulla base di dati in forma di tabella**. La sua forma elementare è:

```
select  Attributo,...,Attributo  
from    Tabella  
where   Condizione
```



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

Istruzione **select** (versione elementare): semantica

La semantica di

select	<i>Attributo,..., Attributo</i>
from	<i>Tabella</i>
where	<i>Condizione</i>

si può descrivere così: ogni tupla t della tabella il cui nome *Tabella* è indicato nella clausola **from** viene analizzata. Se t non soddisfa la condizione nella clausola **where**, allora viene ignorata. Altrimenti da t viene prodotta la target list secondo quanto specificato nella target list che appare dopo **select** e la tupla risultante da tale target list viene inserita nel risultato.

Il risultato della esecuzione della query (la tabella che contiene le tuple calcolate) viene restituito nel canale di output del sistema (e riportato all'utente per la visualizzazione). Vedremo successivamente cosa occorre fare per memorizzarlo nella base di dati (ad esempio in una nuova tabella)

Istruzione **select** (versione elementare): semantica

La semantica di

select *Attributo ... Attributo*
from *Tabella*
where *Condizione*

che abbiamo descritto chiarisce che l'istruzione è **analoga** alla seguente espressione dell'algebra relazionale

$\text{PROJ}_{\text{Attributo}, \dots, \text{Attributo}}(\text{SEL}_{\text{Condizione}}(\text{Tabella}))$

Perché diciamo che è “analoga” e non “equivalente”? Perché in SQL la tabella ed il risultato possono contenere duplicati, mentre nel modello relazionale le relazioni sono insiemi (non multiinsiemi) e quindi il risultato di una espressione dell'algebra, essendo una relazione, è anch'essa **sempre** un insieme di tuple. Ne segue che l'unica cosa che possiamo asserire è che, per ogni base di dati, l'istruzione **select** di SQL fornisce lo stesso risultato dell'espressione dell'algebra relazionale a meno dei duplicati che può contenere. È in questo senso che usiamo il termine “analogo”.

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

Lo schema di questa base di dati è S

persone

<u>nome</u>	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

*nelle slides che seguono
assumiamo che persone
diverse abbiano nomi
diversi*

Selezione e proiezione

Vogliamo nome e reddito delle persone con meno di 30 anni.

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

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

Qui e nelle prossime slides, per le query SQL mostriamo talvolta anche le “analoghe” espressioni dell'algebra relazionale

nome	reddito
Andrea	21
Aldo	15
Filippo	30

Selezione e proiezione

Vogliamo nome e reddito delle persone con meno di 30 anni.

Da ora in poi assumiamo di essere nell'ambito dello schema S (che quindi è implicito) e quindi possiamo omettere il nome della schema. In questo caso possiamo quindi scrivere:

nome	reddito
Andrea	21
Aldo	15
Filippo	30

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

Ricordiamo le convenzioni sui nomi

Nella query che vediamo qui sotto non c'è ambiguità su quali sono gli attributi: essi sono certamente quelli della tabella “persone”. Quindi la query

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

si può scrivere anche come:

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

SELECT: “as” per ridenominazione

“**as**” nella lista degli attributi serve a ridenominare gli attributi, specificando esplicitamente un nome per un attributo del risultato. Quando per un attributo manca tale ridenominazione, il nome dell’attributo nel risultato sarà uguale a quello che compare nella tabella menzionata nella clausola **from**.

Esempio:

```
select nome as name, reddito as salary
from persone
where eta < 30
```

restituisce come risultato una tabella con due attributi, il primo di nome **name** ed il secondo di nome **salary**

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

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

SELECT: “as” per alias

“as” serve anche ad assegnare un nuovo nome (alias) alle tabelle nell’ambito di una query. Ad esempio:

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

ridenominazione

si può scrivere anche:

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

ridenominazione

o anche:

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


SELECT: “as” per alias

“as” serve anche ad assegnare un nuovo nome (alias) alle tabelle nell’ambito di una query. Ad esempio:

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

ridenominazione

si può scrivere anche:

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

alias

o anche:

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

Nota: “as” si
può anche omettere

Proiezione in SQL

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 ai duplicati

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

cognome	filiale
Neri	Napoli
Neri	Milano
Rossi	Roma
Rossi	Roma

senza "distinct":
con duplicati

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

cognome	filiale
Neri	Napoli
Neri	Milano
Rossi	Roma

con "distinct":
senza duplicati

Selezione senza proiezione

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

SEL_{eta<30}(persone)

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

dammi tutti gli
attributi

è un'abbreviazione per:

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

tutti gli
attributi

SELECT con asterisco

Data una tabella R sugli attributi A_1, \dots, A_n

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

equivale a

```
select   $A_1, \dots, A_n$   
from    R  
where   cond
```



Proiezione senza selezione

Nome e reddito di tutte le persone

PROJ_{nome, reddito}(persone)

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

è un'abbreviazione per:

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

Condizione complessa nella clausola “where”

Fino ad ora abbiamo usato espressioni semplici (ossia atomiche, formate da una sola condizione elementare). Ovviamente, però, nella clausola where possono comparire espressioni booleane qualunque, semplici o complesse, ossia formate con i classici operatori booleani ed eventualmente parentesi. Ad esempio:

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

Condizioni con operatore “LIKE”

Nelle condizioni che compaiono nella clausola `where` si possono usare **molte operatori che SQL mette a disposizione**. Rimandiamo al manuale del linguaggio per avere un quadro completo di tali operatori.

Menzioniamo qui l'operatore `like` che consente di verificare che una stringa appartenga al linguaggio definito da una espressione regolare. Ad esempio, se vogliamo conoscere quali sono le persone che hanno un nome che inizia per 'A', ha 'd' come terza lettera e può continuare con altri caratteri, scriviamo la query:

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

espressione regolare
 $(\text{'A'} + \Sigma + \text{'d'})^*$
[Σ denota l'alfabeto]

Gestione dei valori nulli – “is null” e “is not null”

Nelle condizioni che compaiono nella clausola *where* si possono usare anche i predicati “is null” e “is not null” per gestire i valori nulli (già visti in algebra relazionale)

Vogliamo le persone la cui età è o potrebbe essere maggiore di 40

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

```
select *  
from   persone  
where  età > 40 or età is null
```

Espressioni nella target list

Fino ad ora abbiamo usato solo nomi di attributi nella target list (quella che appare dopo `select`). In realtà ogni elemento della target list può essere una espressione che fa uso dei valori memorizzati negli attributi delle tuple del risultato. Ad esempio:

*espressione aritmetica che
fa uso del valore dell'attributo
reddito e lo divide per 2*

```
select età, reddito/2
from persone
where nome = 'Luigi'
```

*in assenza di
ridenominazione, il nome
dell'attributo nella tabella
risultato è uguale
all'espressione*

Risultato:

età	reddito/2
50	20

Espressioni nella target list

Nella target list può comparire un numero qualunque di espressioni e all'interno di tali espressioni possono ovviamente comparire anche costanti (nell'esempio precedente abbiamo usato la costante 2). Ulteriore esempio:

*espressione costituita da
una costante di tipo stringa*

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

Risultato:

nomePersona	età	redditoSemestrale
Luigi	50	20

Esercizio 1

Calcolare la tabella ottenuta dalla tabella **persone** ignorando l'attributo età, selezionando solo le persone con reddito tra 20 e 30, aggiungendo un attributo che ha, in ogni tupla, un valore booleano che indica se la persona corrispondente a quella tupla sta sotto i 50 anni o no ed aggiungendone un altro che indica il reddito mensile. Mostrare poi il risultato dell'interrogazione.

persone

<u>nome</u>	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

Soluzione esercizio 1

*espressione
booleana*

```
select nome, reddito, età < 50 as sotto50,  
       reddito/12 as redditoMensile  
from   persone  
where  reddito >= 20 and reddito <= 30
```

Risultato:

nome	reddito	sotto50	redditoMensile
Andrea	21	true	1.75
Filippo	30	true	2.50
Franco	20	false	1,67

Esercizio 2

Calcolare la tabella ottenuta dalla tabella **persone** selezionando solo quelli con età minore di 30 o maggiore di 60, proiettando i dati sugli attributi **nome** e **reddito**, ed aggiungendo un attributo che ha, in ogni tupla, il valore dell'anno in corso (che si assume 2022).

Mostrare il risultato dell'interrogazione

persone

<u>nome</u>	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

Soluzione esercizio 2

```
select nome, reddito, 2022 as annoInCorso
from persone
where età < 30 or età > 60
```

*espressione
costante*

nome	reddito	annoInCorso
Andrea	21	2022
Aldo	21	2022
Filippo	30	2022
Sergio	35	2022
Luisa	87	2022



Selezione, proiezione e join

- Le istruzioni **select** che abbiamo visto finora hanno una sola tabella nella clausola **from** e quindi permettono di realizzare:
 - selezioni
 - proiezioni
 - ridenominazioni
- I **join** (e i prodotti cartesiani) si possono realizzare indicando due o più tabelle nella clausola **from**, separate da virgola

La forma base della select: sintassi e semantica

La forma base delle select in SQL è:

```
select <target list>
from   R1,R2,...,Rn
where  <condizione>
```

lista di attributi

lista di relazioni

La sua semantica si può descrivere semplicemente dicendo che essa è **analog**a all'espressione dell'algebra relazionale:

$\text{PROJ}_{\langle \text{target list} \rangle} (\text{SEL}_{\langle \text{condizione} \rangle} (R1 \times R2 \times \dots \times Rn))$

Ogni tupla t del prodotto cartesiano viene analizzata. Se t non verifica la condizione della clausola WHERE, essa viene ignorata. Se invece t verifica la condizione della clausola WHERE, allora da t viene prodotta la target list secondo la proiezione specificata nella clausola SELECT e la tupla risultante da tale target list viene inserita nel risultato

La forma base della select: sintassi e semantica

Abbiamo appena detto che la semantica di

```
select <target list>  
from   R1,R2,...,Rn  
where  <condizione>
```

si può descrivere come:

PROJ_{<target list>} (**SEL**_{<condizione>} (R1 × R2 × ... × Rn))

Attenzione: questo non significa che il DBMS calcola davvero il prodotto cartesiano di R1,R2,...,Rn!

Significa che il **risultato ottenuto è lo stesso di quello che** si ottiene calcolando prima il prodotto cartesiano delle tabelle nella clausola from, poi eseguendo la selezione sulla base della clausola where e poi eseguendo la proiezione sulla base della clausola select (con **distinct** il tutto avviene eliminando eventuali duplicati). Il vero modo con cui il DBMS giunge al risultato dipende dall'algoritmo interno che usa, algoritmo che farà di tutto per evitare di calcolare il costoso prodotto cartesiano.

SQL e algebra relazionale (1)

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

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

è analoga quindi a :

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

a sua volta equivalente a

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

a sua volta equivalente al Theta-join:

$PROJ_{A1,A4} (R1 JOIN_{A2=A3} R2)$

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

SQL e algebra relazionale (2)

Possono essere necessarie ridenominazioni

- nella target list (per avere nomi di attributi significativi negli attributi del risultato)
- nel prodotto cartesiano (in particolare, introdurre alias consente di riferirsi due o più volte alla stessa tabella)

Esempio:

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

che, come al solito, si scrive anche senza “as”

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

SQL e algebra relazionale: esempio

Date le tabelle: $R1(A1,A2)$ e $R2(A3,A4)$
la query in SQL

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

è equivalente alla query in algebra relazionale:

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

Il self-join in SQL

Come sappiamo già, il self-join è un join in cui la stessa relazione compare sia come operando sinistro sia come operando destro ed è cruciale quando dobbiamo combinare due tuple della stessa relazione. Supponiamo ad esempio di volere le coppie di persone con lo stesso reddito.

persone


<u>nome</u>	eta	reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	21

Il self-join in SQL

Come sappiamo già, il self-join è un join in cui la stessa relazione compare sia come operando sinistro sia come operando destro ed è cruciale quando dobbiamo combinare due tuple della stessa relazione. Supponiamo ad esempio di volere le coppie di persone con lo stesso reddito. **È immediato verificare che le due tuple collegate dalla linea rossa formano una coppia che soddisfa la condizione. Ma come facciamo a combinarle?**

persone

<u>nome</u>	eta	reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	21



Il self-join in SQL

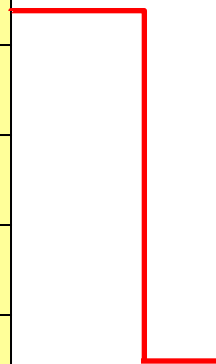
Come sappiamo già, il self-join è un join in cui la stessa relazione compare sia come operando sinistro sia come operando destro ed è cruciale quando dobbiamo combinare due tuple della stessa relazione. Supponiamo ad esempio di volere le coppie di persone con lo stesso reddito. Consideriamo una «copia virtuale» della relazione, ovviamente usando opportuni alias, e usiamo il join per combinare le due tuple collegate dalla linea rossa sulla condizione di uguale reddito.

persone as p1

<u>nome</u>	eta	reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	21

persone as p2

<u>nome</u>	eta	reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	21



Il self-join in SQL

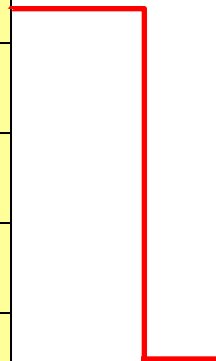
```
select p1.nome, p1.eta, p1.reddito, p2.nome, p2.eta  
from persone p1, persone as p2  
where p1.reddito=p2.reddito
```

persone as p1

<u>nome</u>	eta	reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	21

persone as p2

<u>nome</u>	eta	reddito
Andrea	27	21
Aldo	25	15
Maria	55	42
Anna	50	35
Filippo	26	21



Il self-join in SQL

```
select p1.nome, p1.eta, p1.reddito, p2.nome, p2.eta  
from persone p1, persone as p2  
where p1.reddito=p2.reddito
```

Eseguendo questa query otteniamo:

p1.nome	p1.eta	p1.reddito	p2.nome	p2.eta
Andrea	27	21	Filippo	26
Andrea	27	21	Andrea	27
Filippo	26	21	Andrea	27
Filippo	26	21	Filippo	26
Aldo	25	15	Aldo	25
Maria	55	42	Maria	55
Anna	26	21	Anna	26

Il self-join in SQL

```
select p1.nome, p1.eta, p1.reddito, p2.nome, p2.eta
from persone p1, persone as p2
where p1.reddito=p2.reddito
```

Eseguendo questa query otteniamo:

p1.nome	p1.eta	p1.reddito	p2.nome	p2.eta
Andrea	27	21	Filippo	26
Andrea	27	21	Andrea	27
Filippo	26	21	Andrea	27
Filippo	26	21	Filippo	26
Aldo	25	15	Aldo	25
Maria	55	42	Maria	55
Anna	26	21	Anna	26

non
significative,
perché
chiaramente
ridondanti

Il self-join in SQL

```
select p1.nome, p1.eta, p1.reddito, p2.nome, p2.eta  
from persone p1, persone as p2  
where p1.reddito=p2.reddito and p1.nome<p2.nome
```

Eseguendo questa query otteniamo:

p1.nome	p1.eta	p1.reddito	p2.nome	p2.eta
Andrea	27	21	Filippo	26

lasciando solo
le tuple in cui
p1.nome viene
prima in ordine
alfabetico di
p2.nome
eliminiamo le
tuple non
significative



Altro esempio di self-join in SQL

Data la relazione **Volo(partenza,arrivo)**, ogni tupla della quale rappresenta un volo aereo da una certa città ad un'altra, vogliamo sapere quali sono le città raggiungibili da Roma con due voli.

Altro esempio di self-join in SQL

È facile verificare che la query prevede di trovare due tuple t_1 e t_2 nella relazione Volo tale che $t_1.\text{partenza} = \text{'Roma'}$ e $t_1.\text{arrivo} = t_2.\text{partenza}$. Come abbiamo visto prima, questo si realizza con un self-join.

La query in algebra relazionale sarebbe:

$\text{PROJ}_a(\text{SEL}_{\text{partenza}=\text{'Roma'}}(\text{Volo}) \text{ JOIN}_{\text{arrivo}=\text{p}} \text{ REN}_{\text{p} \leftarrow \text{partenza}, \text{a} \leftarrow \text{arrivo}}(\text{Volo}))$

In SQL la query è:

se non vogliamo
duplicati

```
select distinct V2.arrivo
from   Volo as V1, Volo as V2
where  V1.partenza='Roma' and
       V1.arrivo = V2.partenza
```

SQL: esecuzione delle interrogazioni

- Le espressioni SQL sono dichiarative e noi ne stiamo illustrando la semantica
- In pratica, i DBMS tentano di eseguire le operazioni in modo efficiente, ad esempio:
 - eseguono le selezioni al più presto
 - se possibile, eseguono join e **non** prodotti cartesiani
 - usano strutture ausiliarie, come gli indici
- 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	<u>figlio</u>
Luisa	Maria
Luisa	Luigi
Anna	Olga
Anna	Filippo
Maria	Andrea
Maria	Aldo

paternita

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

persone

<u>nome</u>	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



Assunzioni

Nelle slide che seguono, facciamo queste assunzioni:

- Come abbiamo già detto, persone diverse hanno nomi diversi e non nulli: nome è chiave (primaria)
- Ogni figlio ha un solo padre (figlio è chiave primaria in paternità)
- Ogni figlio ha una sola madre (figlio è chiave primaria in maternità)
- I valori che troviamo nell'attributo nome delle tabelle paternita e maternita si trovano anche nell'attributo nome nella tabella persone (integrità referenziale)
- Se non esplicitamente detto, non ci preoccupiamo di eliminare i duplicati nel risultato delle query e quindi le espressioni dell'algebra relazionale che mostreremo sono analoghe (non necessariamente equivalenti) alle query SQL.



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)

$\text{PROJ}_{\text{padre}}(\text{paternita JOIN}_{\text{figlio=nome}} \text{SEL}_{\text{reddito}>20}(\text{persone}))$

```
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 della quale entrambi i genitori sono noti.

Esprimere la query sia in algebra relazionale sia in SQL.



Esercizio 4: soluzione

Padre e madre di ogni persona della quale entrambi i genitori sono noti.

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” e della quale entrambi i genitori sono noti», allora avremmo dovuto usare un join in più:

In algebra:

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

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 per ognuna il suo nome, il suo reddito e anche il reddito del padre

Esprimere la query sia in algebra relazionale sia in SQL

Esercizio 5: soluzione

Le persone che guadagnano più dei rispettivi padri, mostrando per ognuna il suo nome, il suo reddito e anche il reddito del padre

PROJ_{nome, reddito, RP}
(SEL_{reddito > RP} (REN_{NP, EP, RP} ← nome, eta, reddito (persone)
JOIN_{NP=padre}
(paternita JOIN_{figlio = nome} persone))
)

```
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

In SQL esiste un operatore che si può usare nella clausola **from** e che corrisponde al Theta-join.



Join esplicito

Padre e madre di ogni persona della quale entrambi sono noti.

```
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 per ognuna il suo nome, il suo reddito e anche il 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 per ognuna il suo nome, il suo reddito e anche il 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
```

*due applicazioni
dell'operatore
di join esplicito*

```
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 primo join esplicito nella clausola **from** dà come risultato una tabella con gli attributi: **p.nome**, **p.eta**, **p.reddito**, **t.padre**, **t.figlio**. Questa tabella va in input al secondo join esplicito il cui risultato avrà come attributi: **p.nome**, **p.eta**, **p.reddito**, **t.padre**, **t.figlio**, **f.nome**, **f.eta**, ed **f.reddito**.

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

```
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)

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

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
```

come al solito: equi-join sugli attributi in comune, ovvero attributi che hanno lo stesso nome semplice (il nome semplice è quello ottenuto dal nome esteso ignorando nome di schema e nome di relazione)

Il prodotto cartesiano

È noto che in algebra il join naturale tra due relazioni che non hanno attributi in comune corrisponde al prodotto cartesiano.

Analogamente, in PostgreSQL, il join naturale tra due relazioni che non hanno attributi in comune restituisce il prodotto cartesiano.

Per calcolare il prodotto cartesiano di due relazioni in SQL ci sono anche due altre possibilità:

- esprimere il prodotto cartesiano in modo implicito, mettendo le due relazioni nella lista dopo la clausola from
- esprimere il prodotto cartesiano in modo esplicito, usando l'operatore “cross join” nella clausola from

Join esterno: "outer join"

Padre di ogni persona e, se nota, anche la madre

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

NOTA: "outer" si può anche omettere

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


Join esterno (“outer join”)

- Il **join naturale esterno** (**outer join** in inglese) estende con valori nulli le tuple che non si accoppiano nel join: tali tuple verrebbero tagliate fuori da un join normale ed invece appaiono nel join esterno, accoppiate con valori nulli
- esiste in tre versioni:
 - **join esterno sinistro**: mantiene tutte le tuple del primo operando, estendendole con valori nulli, se necessario
 - **join esterno destro**: mantiene tutte le tuple del primo operando, estendendole con valori nulli, se necessario...
 - **join esterno completo**: mantiene tutte le tuple sia del primo operando e sia del secondo operando, estendendole con valori nulli, se necessario

maternita

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

Padre di ogni persona e, se nota, anche la madre:

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

questa tupla dell'operando sinistro non si combina nel join e quindi compare nell'outer join con il valore NULL negli attributi dell'operando destro

paternita

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

Risultato:

figlio	padre	madre
Franco	Sergio	NULL
Olga	Luigi	Anna
Filippo	Luigi	Anna
Andrea	Franco	Maria
Aldo	Franco	Maria

Outer join, esempi

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

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

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

Outer join, esempi

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

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

*con questo full join non perdo
alcun figlio*

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

Ordinamento del risultato: order by

Nome e reddito delle persone con meno di 30 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	20

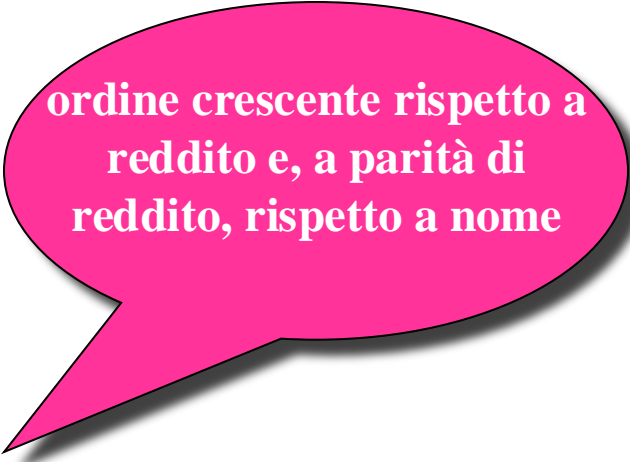
```
select nome, reddito  
from persone  
where eta < 30  
order by nome
```

nome	reddito
Aldo	15
Andrea	21
Filippo	20

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 desc
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 tuple:

– conteggio, minimo, massimo, media, totale

Sintassi base (semplificata):

Funzione ([distinct] EspressioneSuAttributi)

Operatori aggregati: count

Sintassi:

- conta il numero di tuple:

`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 tuple, viene applicato al risultato della seguente interrogazione:

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

Risultato di count: esempio

paternita

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

NumFigliDiFranco
2

count e valori nulli

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

Risultato = numero di tuple
= 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

$(30+36+36)/3$

Operatori aggregati e target list

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

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

L'interrogazione di sopra è irragionevole perché gli elementi della target list sono disomogenei: infatti abbiamo un valore di nome per ogni tupla, mentre abbiamo un valore di max(reddito) per tutta la tabella.

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 di una query
- In molti casi, vorremmo che le funzioni di aggregazione venissero applicate a **gruppi di tuple** delle relazioni
- Per specificare i gruppi di tuple su cui applicare le funzioni, si utilizza la clausola **group by**:

group by *listaAttributi*

Semantica di interrogazioni con operatori aggregati e raggruppamenti

```
select <target list>  
from R  
group by Ai
```

1. Si esegue l'interrogazione **ignorando la group by** e la target list:

```
select *  
from R
```

2. Sulle tuple che risultano si formano i gruppi, dove ogni gruppo si ottiene raggruppando le **tuple che hanno lo stesso valore negli attributi che compaiono nella group by**. Si produce nel risultato una tupla per ogni gruppo e per ognuna di tali tuple si applica la target list, usando ovviamente gli operatori aggregati in essa presenti

Operatori aggregati e raggruppamenti: esempio

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

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

gruppo di
padre='Sergio'

padre	NumFigli
-------	----------

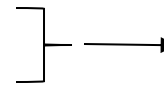
Operatori aggregati e raggruppamenti

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

Operatori aggregati e raggruppamenti

Il numero di figli di ciascun padre

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

gruppo di
padre='Luigi'

paternita

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

padre	NumFigli
Sergio	1

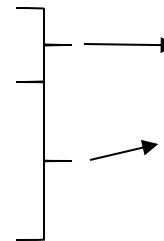
Operatori aggregati e raggruppamenti

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

Operatori aggregati e raggruppamenti

Il numero di figli di ciascun padre

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

gruppo di
padre='Franco'

paternita

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

padre	NumFigli
Sergio	1
Luigi	2

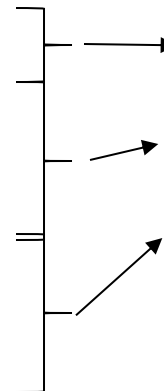
Operatori aggregati e raggruppamenti

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



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

nome	età	reddito
-------------	------------	----------------



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 le persone appartenenti allo 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
```

sembra corretta, perché ogni padre ha un solo reddito,
ma SQL non lo sa e considera disomogenea la target list

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? Dipende dal sistema. PostgreSQL, ad esempio, accetta una target list disomogenea se sa che ogni elemento nella target list ha un unico valore per gruppo (ad esempio, per la presenza di una chiave) , mentre dà errore in caso contrario. Alcuni sistemi, invece, non segnalano errore e restituiscono uno (qualunque) dei valori che sono associati al valore corrente degli attributi che formano il gruppo.

Esempio:

Se volessi avere i redditi delle persone raggruppati per età mediante questa query

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

specificherei, come abbiamo visto prima, una **target list disomogenea**, perché ovviamente possono esistere più valori di reddito per lo stesso gruppo.

In questo caso PostgreSQL dà errore perché in un gruppo di tuple con la stessa età possiamo avere più valori per l'attributo reddito. Al contrario, MySQL, ad esempio, non dà errore e sceglie per ciascun gruppo uno dei valori di reddito che compare nel gruppo, riportandolo nell'elemento "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 (nella clausola **where**), che seleziona le tuple che devono formare i gruppi. Per effettuare la selezione sui gruppi si usa la clausola **having**, che deve apparire dopo la “**group by**” e che di fatto opera un taglio sulle tuple (una per ogni gruppo) del risultato della 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 t.padre, avg(f.reddito)
from   persone f join paternita t
      on f.nome = t.figlio
where  f.eta < 30
group by t.padre
having avg(f.reddito) > 20
```

condizione sulle tuple che
vanno a formare i gruppi

condizione sulle tuple che
rappresentano i gruppi

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 eseguire l'unione

Serve un costrutto esplicito:

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



le target list delle due
select devono avere lo
stesso numero di elementi

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

Differenza

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

le target list delle due
select devono avere lo
stesso numero di elementi

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
```

le target list delle due
select devono avere lo
stesso numero di elementi

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. Ricordiamo com'è la situazione in PostgreSQL
- 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: lo schema totale del database D lo otteniamo giustapponendo tutti gli schemi definiti nell'ambito del database D
- 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 ricordiamo che la notazione estesa per gli oggetti in esso definiti è:
`<nameschema>.<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 tabelle, ovvero
`<nameschema>.<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 (data definition language) 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 corrispondente allo 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

nome
attributo

vincolo

dominio
(o tipo)

vincolo

vincolo

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 (anche più superchiavi per tabella)
 - singolo attributo:
unique dopo la specifica del dominio
 - più attributi:
unique (*Attributo*, ..., *Attributo*)
- **primary key**: definizione della chiave primaria (al massimo una chiave primaria per tabella, su uno o più attributi); sintassi come per la unique, ma usando il termine **primary key**. Ricordiamo che implica **not null**. Osservazione importante: **è cura di chi definisce il vincolo di chiave primaria per K assicurarsi che la superchiave K sia minimale, ossia che non esista una parte di K che è a sua volta una superchiave in tutte le istanze.**
- **check**, per vincoli di tupla o anche più complessi (vedi dopo)

Vincoli intrarelazionali, esempi

```
create table Impiegato (  
    Matricola character(6) constraint pk1 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),  
    constraint un1 unique (Cognome, Nome)  
)
```

nome del
vincolo

nome del
vincolo

primary key, alternative

```
create table Impiegato (  
    Matricola character(6) constraint pk1 primary key,  
    ...  
)
```

oppure

```
create table Impiegato (  
    Matricola character(6),  
    ...  
    constraint pk1 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 ovviamente **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)
```

Nella `alter table` si possono cambiare (`change`, `modify`) o eliminare (`drop`) elementi presenti o aggiungere (`add`) nuovi elementi.

Modifiche degli schemi: **alter table**

La **alter table** è importante anche per potere definire tabelle in cui sono definiti vincoli di integrità referenziali 'incrociati'.

Se per R1 deve essere definito un vincolo di integrità referenziale verso R2 e, viceversa, per R2 deve essere definito un vincolo di integrità referenziale verso R1, sussiste un problema: siccome all'interno della definizione di R1 deve essere menzionato R2, questo vuol dire che per definire R1 deve essere già definita la tabella R2. Ma allo stesso tempo, siccome all'interno della definizione di R2 deve essere menzionato R1, questo vuol dire che per definire R2 deve essere già definita la tabella R1. Ovviamente queste due esigenze sono inconciliabili.

La soluzione è allora questa:

- si definisce R1 senza inserire il vincolo di integrità referenziale verso R2
- si fornisce la definizione completa di R2, quindi anche con il vincolo di integrità referenziale verso R1 (ora già definita)
- con il comando **alter table** si aggiunge alla definizione di R1 il vincolo di integrità referenziale verso R2 (ormai già definita)



Un'importante applicazione di `alter table`

Vogliamo definire due tabelle: `Persona(cf,cittaNascita)`, `Citta(nome,sindaco)` con un vincolo di integrità referenziale da `cittaNascita` a `Citta[nome]` ed uno da `sindaco` a `Persona[CF]`. Tentiamo di procedere così:

```
create table persona(  
  cf varchar(30) primary key,  
  cittaNascita varchar(30)) references citta;
```

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

Un'importante applicazione di `alter table`

Vogliamo definire due tabelle: `Persona(cf,cittaNascita)`, `Citta(nome,sindaco)` con un vincolo di integrità referenziale da `cittaNascita` a `Citta[nome]` ed uno da sindaco a `Persona[CF]`. Tentiamo di procedere così:

```
create table persona(  
  cf varchar(30) primary key,  
  cittaNascita varchar(30)) references citta;
```

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

**errore: la
tabella citta
non è definita**

Un'importante applicazione di `alter table`

Vogliamo definire due tabelle: `Persona(cf,cittaNascita)`, `Citta(nome,sindaco)` con un vincolo di integrità referenziale da `cittaNascita` a `Citta[nome]` ed uno da sindaco a `Persona[CF]`. Procediamo allora così:

```
create table persona(  
  cf varchar(30) primary key,  
  cittaNascita varchar(30)); -- vincolo su cittaNascita non definito
```

```
create table citta(  
  nome varchar(30) primary key,  
  sindaco varchar(100)  
  constraint vincolo1 foreign key (sindaco) references persona);  
-- vincolo definito su sindaco
```

Ora che abbiamo definito `citta` possiamo aggiungere il vincolo su `cittaNascita` mediante “`alter table`”:

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

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 (con vincoli di integrità referenziali)

```
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 “meta dati”, ossia dati relativi a:

- **tabelle**
- **attributi**
- **altri elementi della base di dati e del suo schema**

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

- **Column_Name**
- **Table_name**
- **Ordinal_Position**
- **Column_Default**
- ...

Esempio (ipotetico):

select Column_Name from Columns where Table:Name = 'impiegato'

Si rimanda al manuale per i modi in cui si possono consultare i meta dati in PostgreSQL.



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 tuple di una tabella
- sulla base di una condizione che può coinvolgere anche altre tabelle

Inserimento: sintassi

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

**inserimento di
tuple con i valori
specificati**

oppure

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

**inserimento delle tuple
che sono il risultato
di una select**



Inserimento: esempi

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

```
insert into persone values ('Mario',25,52), ('Anna',30,40)
```

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

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

```
insert into persone (nome)  
select padre  
from paternita  
where padre != 'Giovanni'
```



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 tabella, secondo l'ordine con cui sono stati definiti nella «create table»
- se la lista di attributi non contiene tutti gli attributi della tabella, per gli altri viene inserito il valore di default o, se il valore di default non è specificato, il valore nullo (che deve essere permesso)



Eliminazione di tuple

Sintassi:

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

Esempi:

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

```
delete from paternita  
where figlio != 'Paolo'
```



Eliminazione: commenti

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

Modifica di tuple

- **Sintassi:**

update *NomeTabella*

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

- **Semantica:** vengono modificate le tuple della tabella che soddisfano la condizione “where” secondo quanto stabilito dalla clausola “set”

- *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 (in particolare nella **where**) 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: semantica

La semantica di una «select» che nella clausola «where» ha una query annidata non cambia rispetto a quanto già detto: vengono analizzate le tuple del prodotto cartesiano delle tabelle nella «from» e per ognuna di esse viene valutata l'espressione booleana nella «where». **Quello che cambia è che adesso per valutare l'espressione booleana nella «where» occorre valutare la «select» annidata!** Quindi possiamo assumere che la query annidata venga eseguita per ogni tupla del prodotto cartesiano delle tabelle nella «from» (anche se il motore SQL tenterà di ottimizzare il processo).

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

per ogni tupla di persone viene valutata la clausola «where» e questo comporta l'esecuzione di questa query per ognuna delle tuple di persone

Interrogazioni annidate: operatori

Il risultato di una interrogazione annidata può essere oggetto di confronto nella clausola **where** mediante diversi **operatori**: ad esempio, uguaglianza, disuguaglianza, altri operatori di confronto, ecc. In questo caso il risultato della interrogazione annidata deve essere costituito al massimo da un elemento, tenendo presente che se il risultato è vuoto la condizione nella clausola **where** sarà considerata falsa.

Se non si è sicuri che il risultato sia costituito al massimo da un elemento, si può far precedere l'interrogazione annidata da:

- **any**: la condizione nella clausola **where** sarà vera se la valutazione della espressione di confronto restituisce “true” per **almeno una** delle tuple risultato dell'interrogazione annidate
- **all**: la condizione nella clausola **where** sarà vera se la valutazione della espressione di confronto restituisce “true” per **tutte** le tuple risultato dell'interrogazione annidata

Si può usare

- 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 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 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

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)
               )
```

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 **p.nome** si riferisce alla tabella nella clausola **from** esterna.



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: semantica e visibilità

- **Semantica**: abbiamo già detto che ogni interrogazione interna alla clausola “where” viene eseguita una volta **per ciascuna tupla** nel risultato dell’interrogazione esterna
- Per decidere cosa è visibile nella interrogazione esterna e in quelle interne, vale la classica regola di **visibilità** dei linguaggi di programmazione:
 - Un oggetto di nome X è visibile in una “select” B se X è definito in B oppure se X è (ricorsivamente) visibile nella “select” in cui B è definita, a meno che X non sia mascherata, ossia in B sia definito un oggetto 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 oggetti di uguale nome. Ovviamente, se un nome di oggetto (o tabella) è omesso, si assume il riferimento all’oggetto (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)
```

si riferisce a

si riferisce a

L'attributo **nome** si riferisce alla tabella **persone** nella clausola **from** più vicina.

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 (alias) 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 p join persona f on p.figlio = f.nome
     where z.padre = p.padre and f.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. Usare una query annidata e correlata.

Esercizio 10: soluzione

Nome ed età delle madri che hanno almeno un figlio la cui età differisce meno di 20 anni dalla loro. Usare una query annidata e correlata.

```
select p.nome, p.eta
from   persone p, maternita m
where  p.nome = m.madre and
       m.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)
```

ma non:

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

Le funzioni aggregate possono comparire solo nella `select`, non nella `where`, perché la condizione `where` viene valutata per ogni tupla risultante dalla `from` e applicare una funzione aggregata ad una singola tupla non ha senso

Interrogazioni annidate: condizione su più attributi

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

Quando vogliamo denotare una tupla dobbiamo specificare la lista dei suoi elementi separati da virgola e la lista deve essere racchiusa in parentesi tonde

```
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 la solita, con la differenza che 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) dei vari studenti.

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

```
select miavista.matricola, 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 la solita: per ogni tupla selezionata della tabella calcolata nella clausola from e selezionata dalla clausola where, viene calcolata la target list, e la novità sta nel fatto che questo richiede l'esecuzione di tutte le viste 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**

Clausola CASE

Nella target list può essere utile produrre un valore che dipende da una serie di condizioni. A questo scopo si può usare, negli elementi della target list, la clausola CASE, la cui sintassi è:

```
case
  when <condizione> then <espressione>
  ...
  when <condizione> then <espressione>
  [ else <espressione> ]
end
```

e la semantica è tale per cui l'espressione generata in uscita per un elemento della target list con **case** è

- quella specificata accanto alla prima condizione dopo **when** che viene valutata «vera», *oppure*
- quella accanto alla **else**, se nessuna condizione dopo **when** è vera e la clausola **else** compare, *oppure ancora*
- **null** se nessuna condizione dopo **when** è vera e la clausola **else** non compare.

Clausola CASE

Consideriamo ad esempio la relazione Persona(nome, età, reddito) e supponiamo di volere produrre una relazione formata dalle tuple ottenute da quelle di Persona con età maggiore di 10 aggiungendo un attributo «fasciaeta» i cui possibili valori sono 'ragazzo' (se età ≤ 20), 'adulto' (se età > 20 e ≤ 50), 'anziano' (altrimenti).

La query è:

clausola case

```
select case
    when eta <= 20 then 'ragazzo'
    when eta > 20 and eta <= 50 then 'adulto'
    else 'anziano'
end
    as fasciaeta, reddito
from persona
where eta > 10
```

Vincoli di integrità generici: check

Per specificare vincoli di tupla 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” con query annidata, 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**. Per definire una vista nella base di dati:

```
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 (tuple)
 - **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 temporanee violazioni di vincoli (si veda più avanti il comando SET CONSTRAINTS DEFERRED), in particolare per quei vincoli definiti “deferrable”.
- Se anche una sola 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

Ogni istruzione SQL non eseguita all'interno di una transazione definita esplicitamente è considerata una transazione.

Ma si possono definire transazioni esplicite mediante le seguenti 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

Si noti la differenza tra:

senza transazione:

```
update R set A = 100 where B = 1;
update ContoCorrente
set Saldo = Saldo - (select A from R
                     where B = 1)
where NumeroConto = 12345;

update ContoCorrente
set Saldo = Saldo + (select A from R
                     where B = 1)
where NumeroConto = 55555;
```

A questo punto un altro utente può diminuire il valore di A in R where B = 1 e quindi nel conto 55555 ci vanno meno soldi di quanti si volevano trasferire dal conto 12345

con transazione:

```
begin transaction;
  set transaction isolation level
  serializable;
  update R set A = 100 where B = 1;
  update ContoCorrente
  set Saldo = Saldo - (select A from R
                      where B = 1)
  where NumeroConto = 12345;

  update ContoCorrente
  set Saldo = Saldo + (select A from R
                      where B = 1)
  where NumeroConto = 55555;
end;
```

A questo punto nessun altro utente può cambiare il valore di A in R where B = 1



Vincoli di foreign key: reazioni ad aggiornamenti

SQL dà la possibilità di definire le cosiddette “politiche di reazione alla violazione di vincoli di integrità referenziali”, cioè delle azioni da eseguire quando, aggiornando la base di dati, si violano i vincoli di foreign key.

Nel seguito, se in una tabella T1 è definito un vincolo V di integrità referenziale verso la tabella T2, allora la tabella T1 viene detta tabella “figlia” per il vincolo V e la tabella T2 viene detta tabella “padre” (perché ha tutti i valori della tabella figlia) per V.

Ad esempio, se consideriamo:

```
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));
```

la tabella persona è la tabella “padre” per il vincolo **vincolo1** e la tabella citta è la tabella “figlia” per il vincolo **vincolo1**.

Analogamente, relativamente al vincolo 1, la tupla ('2000','Genova') nella tabella persona viene detta tupla “padre” rispetto alla tupla ('Savona','2000') nella tabella citta, perché ('2000','Genova') è la tupla che contiene nella chiave il valore (cioè 2000) referenziato dalla tupla ('Savona','2000') secondo il vincolo 1. Corrispondentemente, la tupla ('Savona','2000') nella tabella citta viene detta tupla “figlia” rispetto alla tupla ('2000','Genova').

Vincoli di foreign key: reazioni ad aggiornamenti

La specifica di vincolo di foreign key nella tabella T1 (tabella “figlia”) verso la tabella T2 (tabella “padre”) ha questa forma:

FOREIGN KEY (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 padre T2) di una tupla della tabella figlia T1, si genera un errore al momento della verifica del vincolo (dopo l’azione stessa e tutte le azioni ad essa collegate). Si noti che, se il vincolo è deferred, questo momento è la fine della transazione. Si noti anche 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 (prima dell’azione; questo significa che 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/aggiorna anche ogni tupla della tabella T1 che riferenzia la tupla “padre”
- **set default**: quando si cancella/aggiorna la tupla “padre”, si memorizza il valore di default in ogni tupla figlia nell’attributo di T1 definito come foreign key
- **set null**: quando si cancella/aggiorna la tupla “padre”, si memorizza il valore NULL in ogni tupla figlia nell’attributo di T1 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 all’interno della transazione, possiamo dare il comando SET CONSTRAINTS <nome> IMMEDIATE, ma attenzione: quando il sistema esegue questa istruzione, esso controlla i vincoli specificati, senza aspettare la fine della transazione.



Transazioni in SQL: vincoli “deferred”

- Ogni DBMS ha un suo insieme di tipi di vincoli che possono essere definiti come «deferrable»
- In PostgreSQL i vincoli che possono essere definiti come Deferrable sono: Unique, Primary Key, Foreign Key, and Exclude (noi non trattiamo quest'ultimo tipo di vincoli)
- Il tipo più importante è, comunque, il vincolo di foreign key

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 per i due vincoli di integrità referenziale vale il default “**no action**”. 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) eseguiamo l'update delle tuple di persona che referenziano Roma, mettendo NULL al posto di Roma. In questo modo evitiamo che rimangano tuple figlie orfane di padre, ossia tuple che referenziano Roma quando quest'ultimo valore viene eliminato dalla tabella citta
- 2) potevamo aver definito nella create table "persona" il vincolo di foreign key come cascade: in questo modo una volta che cancelliamo la tupla della città di nome 'Roma' vengono cancellate automaticamente le tuple figlie in persona, ossia tuple corrispondenti a persone nate a Roma
- 3) se nella create table "persona" il vincolo di foreign key è definito come "deferrable", allora per eseguire la cancellazione di Roma possiamo usare una transazione dentro la quale dichiariamo il vincolo "deferred", per cui la cancellazione della tupla della città di nome 'Roma' non causa immediatamente una violazione, visto che la verifica del vincolo sarà effettuata alla fine della transazione. Ovviamente, prima della fine della transazione dobbiamo «mettere a posto» le tuple di persona che referenziavano Roma al fine di eliminare le violazioni di vincoli

Esempio: metodo 1)

Per eseguire la cancellazione della città Roma:

```
update persona set cittanascita = null where cittanascita = 'Roma';
```

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

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

Esempio: metodo 2)

Sostituiamo a quelle di prima le seguenti definizioni delle create table, che associano la politica «on delete cascade» al vincolo di foreign key che appare nella tabella persona:

```
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) on delete cascade;
insert into citta values ('Roma',null), ('Milano',null);
insert into persona values ('100','Roma');
```

Ora cancelliamo la tupla relativa alla città di Roma, con un semplice delete:

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

e non otteniamo un errore, ma otteniamo la cancellazione delle tuple relative alle 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';  
<< mettiamo a posto le tuple di persona che referenziano Roma>>;  
end;
```