Basi di dati e programmazione web
Lezione 2

Prof. Paolo Ciaccia
paolo.ciaccia@unibo.it

DEIS – Università degli Studi di Bologna

Obiettivi della lezione

- Completare la descrizione del linguaggio SQL
- Introdurre il concetto di transazione dal punto di vista logico

Linguaggio SQL

Il DB di esempio

Università

Studenti

Matricola	Cognome	Nome	DataNascita	Email
29323	Bianchi	Giorgio	21/06/1978	gbianchi@alma.unibo.it
35467	Rossi	Anna	13/04/1978	anna.rossi@yahoo.it
39654	Verdi	Marco	20/09/1979	mverdi@mv.com
42132	Neri	Lucia	15/02/1978	lucia78@cs.ucsd.edu

Corsi

CodCorso	Titolo	Docente	Anno
483	Analisi	Biondi	1
729	Analisi	Neri	1
913	Sistemi Informativi	Castani	2

Esami

Matricola	CodCorso	Voto	Lode
29323	483	28	NO
39654	729	30	SÌ
29323	913	26	NO
35467	913	30	NO

Interrogazioni: che altro?

- Quanto visto sinora ci consente, per una data relazione, di:
  - Definire il suo schema, con tutti i vincoli opportuni
  - Inserire i dati, modificarli e cancellarli
  - Scrivere delle interrogazioni che operano su tale relazione
- Le interrogazioni "interessanti" includono tuttavia almeno altre 2 tipologie notevoli di casi:
- Interrogazioni su più relazioni

  I docenti dei corsi di cui lo studente con matricola 29323 ha sostenuto l'esame
- Interrogazioni di sintesi

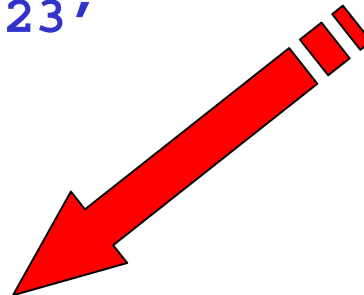
  Quanti esami ha verbalizzato ciascun docente?

Interrogazioni su più tabelle (1)

E se volessimo trovare i docenti dei corsi di cui lo studente con matricola 29323 ha sostenuto l'esame?

```
SELECT   CodCorso
FROM     Esami
WHERE    Matricola = '29323'
```

CodCorso
483
913



```
SELECT   Docente
FROM     Corsi
WHERE    CodCorso IN (483,913)
```

Docente
Biondi
Castani

- E se lo studente avesse sostenuto 20 esami!? Molto poco pratico!!!
- In più c'è un altro problema…

Interrogazioni su più tabelle (2)

E se volessimo generare un elenco con il seguente formato (schema)?

Matricola	Cognome	Nome	CodCorso	Voto	Lode

- Operando su una singola tabella alla volta non ce la faremmo mai!

Interrogazioni su più tabelle (3)

- Per prima cosa cerchiamo di capire cosa conterrebbe il risultato…

Studenti

Matricola	Cognome	Nome	DataNascita	Email
29323	Bianchi	Giorgio	21/06/1978	gbianchi@alma.unibo.it
35467	Rossi	Anna	13/04/1978	anna.rossi@yahoo.it
39654	Verdi	Marco	20/09/1979	mverdi@mv.com
42132	Neri	Lucia	15/02/1978	lucia78@cs.ucsd.edu

Esami

Matricola	CodCorso	Voto	Lode
29323	483	28	NO
39654	729	30	SÌ
29323	913	26	NO
35467	913	30	NO

Matricola	Cognome	Nome	CodCorso	Voto	Lode
29323	Bianchi	Giorgio	483	28	NO
39654	Verdi	Marco	729	30	SÌ
29323	Bianchi	Giorgio	913	26	NO
35467	Rossi	Anna	913	30	NO

Interrogazioni su più tabelle (5)

- Come abbiamo fatto?
  1. Come prima cosa abbiamo preso le tabelle Esami e Studenti
  2. Poi abbiamo "accoppiato" ciascuna tupla di Esami con la corrispondente tupla di Studenti, usando la Matricola
  3. Abbiamo poi mantenuto solo gli attributi che ci interessavano
- I punti 1 e 3 sono semplici da realizzare, e sono molto simili a quello che si fa quando si opera su una relazione sola
- Per il punto 2 dobbiamo esplicitare il "criterio di accoppiamento" che, in termini più tecnici, si chiama condizione di join ("*giunzione*")

Abbiamo eseguito un join di Studenti ed Esami

Matricola	Cognome	Nome	CodCorso	Voto	Lode
29323	Bianchi	Giorgio	483	28	NO
39654	Verdi	Marco	729	30	SÌ
29323	Bianchi	Giorgio	913	26	NO
35467	Rossi	Anna	913	30	NO

Esprimere la condizione di join (1)

- Per il punto 2 dobbiamo esplicitare il "criterio di accoppiamento" che, in termini più tecnici, si chiama condizione di join ("*giunzione*")
- Nel nostro caso il criterio è, a parole

Accoppia una tupla di Esami con una tupla di Studenti se hanno la stessa (uguale) Matricola

- Ma se scriviamo `Matricola = Matricola` non significa nulla!
- Perché? Perché il sistema, quando scrivo `Matricola`, non sa se intendo quella della tabella Studenti o quella della tabella Esami

Esprimere la condizione di join (2)

Laureati

Matricola	VotoFinale
29323	89
35467	95
39654	102

Concorsi

Codice	VotoFinale
ABC	105
XYZ	88
GHJ	99

- In questo caso il criterio è, a parole

Accoppia una tupla di Laureati con una tupla di Concorsi se il VotoFinale del primo è maggiore o uguale al VotoFinale minimo richiesto per l'ammissione al concorso stesso

Matricola	Codice
29323	XYZ
35467	XYZ
39654	XYZ
39654	GHJ

- Ma se scriviamo `VotoFinale >= VotoFinale` non significa nulla!

Indicare correttamente gli attributi

- Quando lavoriamo su 2 o più tabelle che hanno attributi con lo stesso nome, se vogliamo usare tali attributi nelle interrogazioni dobbiamo indicare esplicitamente la loro tabella, in questo modo:

```
Esami.Matricola
```

- Quindi, negli esempi visti, scriveremmo

```
Esami.Matricola = Studenti.Matricola

Laureati.VotoFinale >= Concorsi.VotoFinale
```

- Questa cosa si può sempre fare, anche quando non è necessario (ad es. perché si opera su una sola tabella, o se gli attributi hanno nomi diversi)

Pseudonimi per i nomi delle relazioni

- Se le nostre relazioni hanno nomi lunghi è noioso scriverli per esteso ogni volta
- E' possibile usare degli pseudonimi, o alias, più brevi, che si inseriscono nella clausola FROM (tipicamente l'iniziale del nome, i primi caratteri,ecc.)

```
SELECT ...
FROM   Esami E ...
WHERE  E.Matricola ...
```

- Adesso abbiamo tutto quello che ci serve…

…e finalmente…

1. Come prima cosa prendiamo le tabelle Esami e Studenti

```
FROM     Esami E, Studenti S
```

2. Poi “accoppiamo” ciascuna tupla di Esami con la corrispondente tupla di Studenti, usando la Matricola

```
WHERE    E.Matricola = S.Matricola
```

3. Infine manteniamo solo gli attributi che ci interessano

```
SELECT   S.Matricola,S.Cognome,S.Nome,E.Voto,E.Lode
```

```
SELECT   S.Matricola,S.Cognome,S.Nome,E.Voto,E.Lode
FROM     Esami E, Studenti S
WHERE    E.Matricola = S.Matricola
```

Altri esempi (1)

i numeri di matricola degli studenti
che hanno sostenuto l'esame di Analisi con il Prof. Biondi

CodCorso	Titolo	Docente	Anno
483	Analisi	Biondi	1
729	Analisi	Neri	1
913	Sistemi Informativi	Castani	2

```
SELECT   E.Matricola
FROM     Corsi C, Esami E
WHERE    C.CodCorso = E.CodCorso
AND      C.Titolo = 'Analisi'
AND      C.Docente = 'Biondi'
```

Matricola	CodCorso	Voto	Lode
29323	483	28	NO
39654	729	30	SÌ
29323	913	26	NO
35467	913	30	NO

Matricola	CodCorso	Voto	Lode	Titolo	Docente	Anno
29323	483	28	NO	Analisi	Biondi	1
39654	729	30	SÌ	Analisi	Neri	1
29323	913	26	NO	Sistemi Informativi	Castani	2
35467	913	30	NO	Sistemi Informativi	Castani	2

Altri esempi (2)

i docenti dei corsi di cui lo studente con matricola 29323 ha sostenuto l'esame

```
SELECT   C.Docente
FROM     Corsi C, Esami E
WHERE    C.CodCorso = E.CodCorso
AND      E.Matricola = '29323'
```

CodCorso	Titolo	Docente	Anno
483	Analisi	Biondi	1
729	Analisi	Neri	1
913	Sistemi Informativi	Castani	2

Matricola	CodCorso	Voto	Lode
29323	483	28	NO
39654	729	30	SÌ
29323	913	26	NO
35467	913	30	NO

Matricola	CodCorso	Voto	Lode	Titolo	Docente	Anno
29323	483	28	NO	Analisi	Biondi	1
~~39654~~	~~729~~	~~30~~	~~SÌ~~	~~Analisi~~	~~Neri~~	~~1~~
29323	913	26	NO	Sistemi Informativi	Castani	2
~~35467~~	~~913~~	~~30~~	~~NO~~	~~Sistemi Informativi~~	~~Castani~~	~~2~~

Più di 1 tabella = 2,3,4,…

- Quanto fatto con 2 tabelle si può generalizzare al caso di 3 o più tabelle

i docenti dei corsi di cui lo studente Giorgio Bianchi ha sostenuto l'esame

```
SELECT C.Docente
FROM   Corsi C, Esami E, Studenti S
WHERE  C.CodCorso = E.CodCorso
AND    E.Matricola = S.Matricola
AND    S.Cognome = 'Bianchi'
AND    S.Nome = 'Giorgio'
```

Self Join

- In alcuni casi è necessario fare il join di una tabella con se stessa
- Essenziale fare uso di alias

Chi sono i nonni di Anna?

Genitori G1

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

Genitori G2

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

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

Nonno
Giorgio
Silvia
Enzo

Join espliciti

- Anziché scrivere i predicati di join nella clausola WHERE è possibile definire una *join table* nella clausola FROM

```
SELECT   S.*, E.CodCorso, E.Voto, E.Lode
FROM     Studenti S JOIN Esami E
                  ON (S.Matricola = E.Matricola)
WHERE    E.Voto > 26
```

in cui JOIN si può anche scrivere INNER JOIN

- Altri tipi di join espliciti sono:

```
LEFT [OUTER] JOIN
RIGHT [OUTER] JOIN
FULL [OUTER] JOIN
NATURAL [LEFT|RIGHT [OUTER]] JOIN
```

Outer join

- L'outer join è un tipo di join che permette di restituire anche le tuple che non soddisfano mai la condizione di join

```
SELECT   *
FROM     Studenti S LEFT JOIN Esami E
                    ON (S.Matricola = E.Matricola)
```

restituisce anche gli studenti (left operand) senza esami, quindi tutti

- Pertanto

```
SELECT   *
FROM     Studenti S LEFT JOIN Esami E
                    ON (S.Matricola = E.Matricola)
WHERE E.Voto IS NULL       -- oppure E.CodCorso, ecc.
```

trova gli studenti senza esami

Operatori insiemistici

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

`UNION, INTERSECT, EXCEPT`

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

`UNION ALL, INTERSECT ALL, EXCEPT ALL`

Operatori insiemistici: esempi (1)

R

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

S

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

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

1
2
3
4

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

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

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

A
1
2
3
4

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

Non corretta!

Operatori insiemistici: esempi (2)

R

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

S

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

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

B
a
b
c

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

B
d

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

B
a
a
b
c

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

B
a
b

Informazioni di sintesi

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

  Esempio: gli esami dello studente con matricola 29323, i nomi degli studenti che hanno sostenuto un esame con il prof. Biondi, ecc.

- In molti casi è viceversa utile ottenere dal DB informazioni (di sintesi) che caratterizzano "gruppi" di tuple

  Esempio: il numero di esami sostenuti dallo studente con matricola 29323, la media dei voti degli esami del primo anno, ecc.

- A tale scopo SQL mette a disposizione due strumenti di base:
  - Funzioni aggregate
  - Clausola di raggruppamento (GROUP BY)

Un nuovo DB per gli esempi…

Imp

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

Sedi

Sede	Responsabile	Citta
S01	Biondi	Milano
S02	Mori	Bologna
S03	Fulvi	Milano

Prog

CodProg	Citta
P01	Milano
P01	Bologna
P02	Bologna

Funzioni aggregate (1)

- Lo standard SQL mette a disposizione una serie di funzioni aggregate (o "di colonna"):
  - MIN minimo
  - MAX massimo
  - SUM somma
  - AVG media aritmetica
  - STDEV deviazione standard
  - VARIANCE varianza
  - COUNT contatore

```
SELECT   SUM(Stipendio) AS ToTStipS01
FROM     Imp
WHERE    Sede = 'S01'
```

TotStipS01
5100

Funzioni aggregate (2)

- L'argomento di una funzione aggregata è una qualunque espressione che può figurare nella SELECT list (ma non un'altra funzione aggregata!)

```
SELECT   SUM(Stipendio*12) AS ToTStipAnnuiS01
FROM     Imp
WHERE    Sede = 'S01'
```

TotStipAnnuiS01
61200

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

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

4100

COUNT e valori nulli

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

Imp

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

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

NumImpS01
4

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

NumStipS01
3

Funzioni aggregate e tipo del risultato

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

Imp

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

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

AvgStip
1412

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

AvgStip
1412.50

Clausola SELECT e funzioni aggregate

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

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

non va bene!
(viceversa, `SELECT MIN(Stipendio), MAX(Stipendio)..` è corretto)

- Il motivo è che una funzione aggregata restituisce un singolo valore, mentre il riferimento a una colonna è in generale un insieme di valori (eventualmente ripetuti)

Funzioni aggregate e raggruppamento

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

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

Sede	NumProg
S01	2
S03	1
S02	1

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

Come si ragiona con il GROUP BY

- Le tuple che soddisfano la clausola WHERE…

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

- …sono raggruppate per valori uguali della/e colonna/e presenti nella clausola GROUP BY…

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

- …e infine a ciascun gruppo si applica la funzione aggregata

Sede	NumProg
S01	2
S03	1
S02	1

GROUP BY: esempi

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

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

Ruolo	AvgStip
Analista	2000
Sistemista	1100
Programmatore	1000

2) Per ogni sede di Milano, lo stipendio medio

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

Sede	AvgStip
S01	1275
S03	1000

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

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

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

Raggruppamento e proiezione

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

```
SELECT   Sede
FROM     Imp
GROUP BY Sede
```

Sede
S01
S02
S03

equivale pertanto a

```
SELECT   DISTINCT Sede
FROM     Imp
```

Condizioni sui gruppi

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

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

Sede	NumImp
S01	4
S02	3

- La clausola HAVING ha per i gruppi una funzione simile a quella che la clausola WHERE ha per le tuple

Tipi di condizioni sui gruppi

- Nella clausola HAVING si possono avere due tipi di condizioni:
  - Condizioni che fanno uso di funzioni aggregate (es. `COUNT(*) > 2`)
  - Condizioni che si riferiscono alle colonne di raggruppamento
    - Queste ultime si possono anche inserire nella clausola WHERE

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

equivale a

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

Sede	NumImp
S02	3
S03	1

Un esempio completo

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

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

L'ordine delle clausole è sempre come nell'esempio

Si ricordi che il GROUP BY non implica alcun ordinamento del risultato

Subquery

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

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

Sede
S01
S03

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

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

Subquery scalari

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

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

- La presenza di vincoli può essere sfruttata a tale scopo

```
SELECT    Responsabile
FROM      Sedi
WHERE     Sede = (SELECT Sede -- al massimo una sede
                  FROM   Imp
                  WHERE  CodImp = ‘E001’)
```

Subquery: caso generale

- Se la subquery può restituire più di un valore si devono usare le forme
  - <op> ANY: la relazione <op> vale per almeno uno dei valori
  - <op> ALL : la relazione <op> vale per tutti i valori

```
SELECT   Responsabile
FROM     Sedi
WHERE    Sede = ANY (SELECT   Sede
                     FROM     Imp
                     WHERE    Stipendio > 1500)
```

```
SELECT   CodImp    -- impiegati con stipendio minimo
FROM     Imp
WHERE    Stipendio <= ALL (SELECT   Stipendio
                           FROM     Imp)
```

- La forma = ANY equivale a IN

Subquery: livelli multipli di innestamento

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

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

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

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

Subquery: quantificatore esistenziale

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

```
SELECT    Sede
FROM      Sedi S
WHERE     EXISTS  (SELECT   *
                   FROM     Imp
                   WHERE    Ruolo = 'Programmatore')
```

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

Subquery correlate

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

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

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

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

Subquery: “unnesting” (1)

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

```
SELECT   DISTINCT Sede
FROM     Sedi S, Imp I
WHERE    S.Sede = I.Sede
  AND    I.Ruolo = ‘Programmatore’
```

- Si noti la presenza del DISTINCT
- La forma innestata è “più procedurale” di quella piatta e, a seconda dei casi, può risultare più semplice da derivare

Subquery: “unnesting” (2)

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

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

- È facile sbagliare, ad esempio la seguente query non è corretta

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

perché la clausola WHERE non è mai soddisfatta!

Subquery: come eseguire la "divisione"

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

  *Sedi in cui sono presenti **tutti** i ruoli*

  equivale a *Sedi in cui non esiste un ruolo non presente*

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

- Il blocco più interno viene valutato per ogni combinazione di S e I1
- Il blocco intermedio funge da "divisore" (interessa I1.Ruolo)
- Data una sede S, se in S manca un ruolo:
  - la subquery più interna non restituisce nulla
  - quindi la subquery intermedia restituisce almeno una tupla
  - quindi la clausola WHERE non è soddisfatta per S

Divisione: esercizio

Voli

Codice	Data
AZ427	21/07/2001
AZ427	23/07/2001
AZ427	24/07/2001
TW056	21/07/2001
TW056	24/07/2001
TW056	25/07/2001

Linee

Codice
AZ427
TW056

Data
21/07/2001
24/07/2001

Trovare le date con voli per tutte le linee

In generale, la divisione è utile per interrogazioni di tipo "universale"

- Gli studenti che hanno dato tutti gli esami del primo anno

Subquery: aggiornamento dei dati

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

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

UPDATE Imp
SET    Stipendio = 1.1*Stipendio
WHERE  Sede IN (SELECT S.Sede
                FROM   Sede S, Prog P
                WHERE  S.Citta = P.Citta
                  AND  P.CodProg = 'P02')
```

Subquery e CHECK

- Facendo uso di subquery nella clausola CHECK è possibile esprimere vincoli arbitrariamente complessi (in teoria…☹)

Ogni sede deve avere almeno due programmatori

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

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

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

Definizione di viste

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

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

ProgSedi

CodProg	CodSede
P01	S01
P01	S03
P01	S02
P02	S02

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

CodProg	CodSede
P01	S01
P01	S03
P01	S02

Uso delle viste

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

Indipendenza logica tramite VIEW

- A titolo esemplificativo si consideri un DB che contiene la tabella

  EsamiBD(Matr,Cognome,Nome,DataProva,Voto)

- Per evitare di ripetere i dati anagrafici e per tenera anche traccia dell'eventuale verbalizzazione del voto (SI/NO), si decide di modificare lo schema del DB sostituendo alla tabella **EsamiBD** le due seguenti:

  StudentiBD(Matr,Cognome,Nome,Registrato)

  ProveBD(Matr,DataProva,Voto)

- È possibile ripristinare la "visione originale" in questo modo:

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

Query complesse che usano VIEW (1)

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

  La sede che ha il massimo numero di impiegati

- La soluzione senza viste è:

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

Query complesse che usano VIEW (2)

- La soluzione con viste è:

```
CREATE VIEW NumImp(Sede,Nimp)
AS      SELECT   Sede, COUNT(*)
        FROM     Imp
        GROUP BY Sede

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

NumImp

Sede	NImp
S01	4
S02	3
S03	1

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

Aggiornamento di viste

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

```
CREATE VIEW NumImpSedi(Sede,NumImp)
AS       SELECT    Sede,COUNT(*)
         FROM      Imp
         GROUP BY  Sede

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

NumImpSedi

Sede	NumImp
S01	4
S02	3
S03	1

- Cosa significa? Non si può fare!
- In generale, ogni DBMS pone dei limiti su quelle che sono le viste aggiornabili

Aggiornabilità di viste (1)

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

Aggiornabilità di viste (2)

- In alcuni casi è solo il blocco più esterno della query di definizione che non deve contenere dei join. Ad esempio, la seguente vista non è aggiornabile

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

mentre lo è questa, di fatto equivalente alla prima

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

Viste con CHECK OPTION

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

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

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

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

Tipi di CHECK OPTION

- Se la vista V1 è definita in termini di un'altra vista V2, e si specifica la clausola **`WITH CHECK OPTION`**, il DBMS verifica che la nuova tupla t inserita soddisfi sia la definizione di V1 che quella di V2, indipendentemente dal fatto che V2 sia stata a sua volta definita **`WITH CHECK OPTION`**

- Questo comportamento di default, che è equivalente a definire V1

  `WITH CASCADED CHECK OPTION`

  si può alterare definendo V1

  `WITH LOCAL CHECK OPTION`

- In modalità **`LOCAL`**, il DBMS verifica solo che t soddisfi la specifica di V1 e quelle di tutte e sole le viste da cui V1 dipende per cui è stata specificata la clausola **`WITH CHECK OPTION`**

Esempio

```
CREATE TABLE R (A INT)   CREATE VIEW V1
                         AS SELECT * FROM R WHERE A < 2
                         WITH CHECK OPTION

CREATE VIEW V2
AS      SELECT * FROM V1 WHERE A > 0
WITH LOCAL CHECK OPTION

CREATE VIEW V3
AS      SELECT * FROM V1 WHERE A > 0
WITH CASCADED CHECK OPTION

INSERT INTO V2(2)   -- OK
INSERT INTO V3(2)   -- Errore su V1
```

Table expressions

- Tra le caratteristiche più interessanti di SQL vi è la possibilità di usare all'interno della clausola FROM una subquery che definisce "dinamicamente" una tabella derivata, e che qui viene anche detta "table expression" ("derived table" in MySQL)

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

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

SM

Sede	MaxStip
S01	2000
S02	2500
S03	1000

Table expressions correlate (1)

- Una table expression può essere correlata a un'altra tabella nella clausola FROM (verificare in MySQL…)

Per ogni sede, la somma degli stipendi pagati agli analisti

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

- Si noti che sedi senza analisti compaiono in output con valore nullo per `TotStip`. Usando il GROUP BY lo stesso risultato si potrebbe ottenere con un LEFT OUTER JOIN, ma occorre fare attenzione…

Table expressions correlate (2)

Per ogni sede, il numero di analisti e la somma degli stipendi ad essi pagati

```
SELECT S.Sede, Stip.NumAn, Stip.TotStip
FROM   Sedi S,
       (SELECT COUNT(*) AS NumAn, SUM(Stipendio) AS TotStip
             FROM Imp I
             WHERE I.Sede = S.Sede
             AND I.Ruolo = 'Analista') AS Stip
```

- Per sedi senza analisti `NumAn` vale 0 e `TotStip` è nullo. Viceversa

```
SELECT   S.Sede,COUNT(*) AS NumAn,SUM(Stipendio) AS TotStip
FROM     Sedi S LEFT OUTER JOIN Imp I
             ON (I.Sede = S.Sede) AND (I.Ruolo = 'Analista')
GROUP BY S.Sede
```

ha per le sedi senza analisti `TotStip` nullo, ma `NumAn` pari a 1!! (in quanto per ognuna di tali sedi c'è una tupla nel risultato dell'outer join). È quindi necessario usare, ad esempio, `COUNT(CodImp)`

Limiti delle table expressions

- Si consideri la query

  La sede in cui la somma degli stipendi è massima
- La soluzione con table expressions è

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

- Benché la query sia corretta, non viene sfruttato il fatto che le due table expressions sono identiche, il che porta a una valutazione inefficiente e a una formulazione poco leggibile

Common table expressions

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

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

WITH e interrogazioni ricorsive (1)

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

  Trova tutti gli antenati (genitori, nonni, bisnonni,…) di Anna
- La query è ricorsiva (non è esprimibile in algebra relazionale, in quanto richiede un numero di (self-)join non noto a priori)
- La formulazione mediante common table expressions definisce la vista temporanea (ricorsiva) `Antenati(Persona,Avo)` facendo l'unione di:
  - una "subquery base" non ricorsiva (che inizializza `Antenati` con le tuple di `Genitori`)
  - una "subquery ricorsiva" che ad ogni iterazione aggiunge ad `Antenati` le tuple che risultano dal join tra `Genitori` e `Antenati`

Genitori

Figlio	Genitore
Anna	Luca
Luca	Maria
Luca	Giorgio
Giorgio	Lucia

Antenati

Persona	Avo
Anna	Luca
Luca	Maria
Luca	Giorgio
Giorgio	Lucia

+

Antenati

Persona	Avo
Anna	Maria
Anna	Giorgio
Luca	Lucia

+

Antenati

Persona	Avo
Anna	Lucia

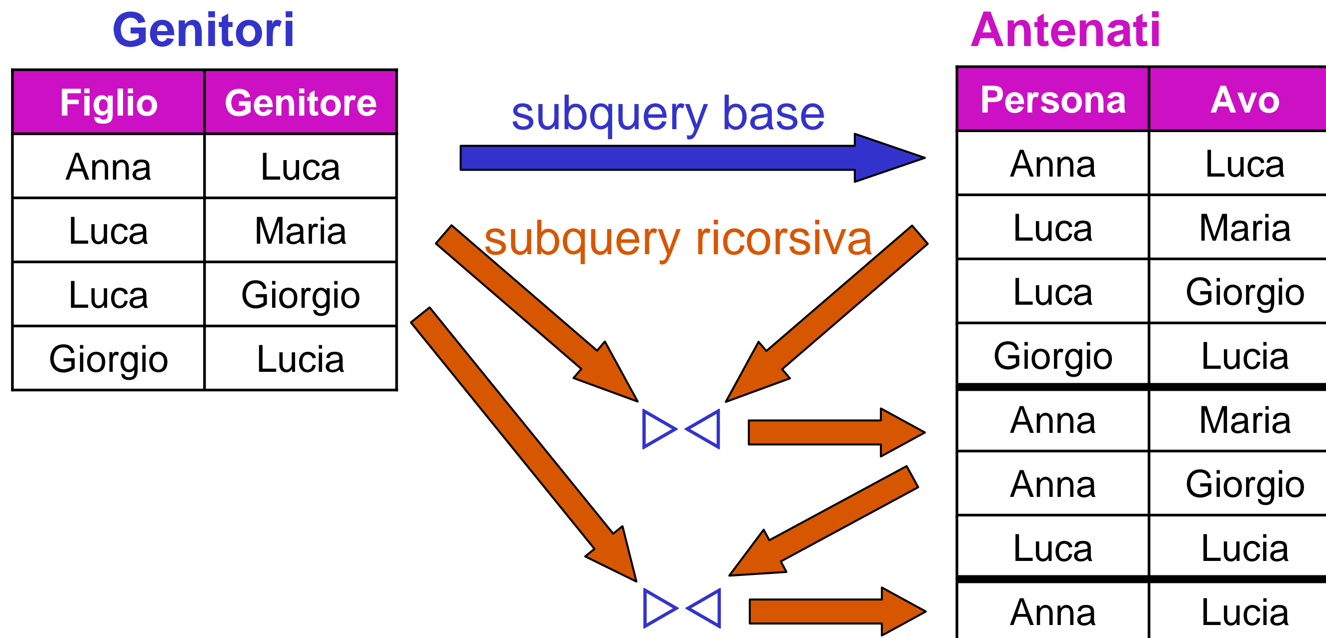
WITH e interrogazioni ricorsive (2)

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

WITH e interrogazioni ricorsive (3)

- Per capire meglio come funziona la valutazione di una query ricorsiva, e come “ci si ferma”, si tenga presente che

  ad ogni iterazione il DBMS aggiunge ad `Antenati` le tuple che risultano dal join tra `Genitori` e **le sole tuple aggiunte ad `Antenati` al passo precedente**



Genitori

Figlio	Genitore
Anna	Luca
Luca	Maria
Luca	Giorgio
Giorgio	Lucia

Antenati

Persona	Avo
Anna	Luca
Luca	Maria
Luca	Giorgio
Giorgio	Lucia
Anna	Maria
Anna	Giorgio
Luca	Lucia
Anna	Lucia

Supporto di SQL nei sistemi F/OS

- Il supporto di SQL nei sistemi F/OS (ma non solo!) è non omogeneo, e varia da una versione all'altra
- Considerando le più recenti versioni dei due sistemi di riferimento (MySQL e PostgreSQL), si può tuttavia asserire che queste sono comparabili per ciò che riguarda le caratteristiche sinora viste, e che le principali differenze vanno invece ricercate in aspetti che si riferiscono a:
  - Organizzazione fisica dei dati (es. partizionamento)
  - Prestazioni (inclusa l'ottimizzazione delle interrogazioni)
  - Tool di amministrazione

MySQL

- A titolo di esempio, si riportano alcune delle caratteristiche di base di SQL che sono state introdotte solo nelle ultime versioni di MySQL
  (fonte: MySQL 5.1 Reference Manual,
  http://downloads.mysql.com/docs/refman-5.1-en.a4.pdf)

Feature	**MySQL Series**
UNION	4.0
Subqueries	4.1
Views	5.0
Foreign keys	5.2 (3.23 per InnoDB tables)

  - InnoDB table: tabella memorizzata in una struttura che permette il supporto di transazioni ACID e il lock a livello di tupla , a fronte di una riduzione di prestazioni
  - Il default è avere table di tipo MyISAM, che garantiscono le migliori prestazioni in lettura, ma rinunciano alle caratteristiche di cui sopra

PostgreSQL

- Per PostgresSQL un elenco come il precedente non è facilmente ottenibile. Per contro, le seguenti sono alcune delle lacune rispetto al "core" dello standard SQL:1999

  (fonte: PostgreSQL 8.2.0 Documentation,

  http://www.postgresql.org/files/documentation/pdf/8.2/postgresql-8.2-A4.pdf)

Feature
Subqueries in CHECK
Views with CHECK OPTION

  - Va tuttavia notato che PostgreSQL mette a disposizione un potente sistema di regole con cui è possibile sopperire (anche se in modo non standard) a tali lacune e a molte altre ancora

Gestione delle transazioni

Cos'è una transazione?

- Una transazione è un'unità logica di elaborazione che corrisponde a una serie di operazioni fisiche elementari (letture/scritture) sul DB
- Esempi:
  - Trasferimento di una somma da un conto corrente ad un altro

```
UPDATE CC                         UPDATE CC
SET    Saldo = Saldo - 50         SET    Saldo = Saldo + 50
WHERE  Conto = 123                WHERE  Conto = 235
```

  - Aggiornamento degli stipendi degli impiegati di una sede

```
UPDATE Imp
SET    Stipendio = 1.1*Stipendio
WHERE  Sede = 'S01'
```

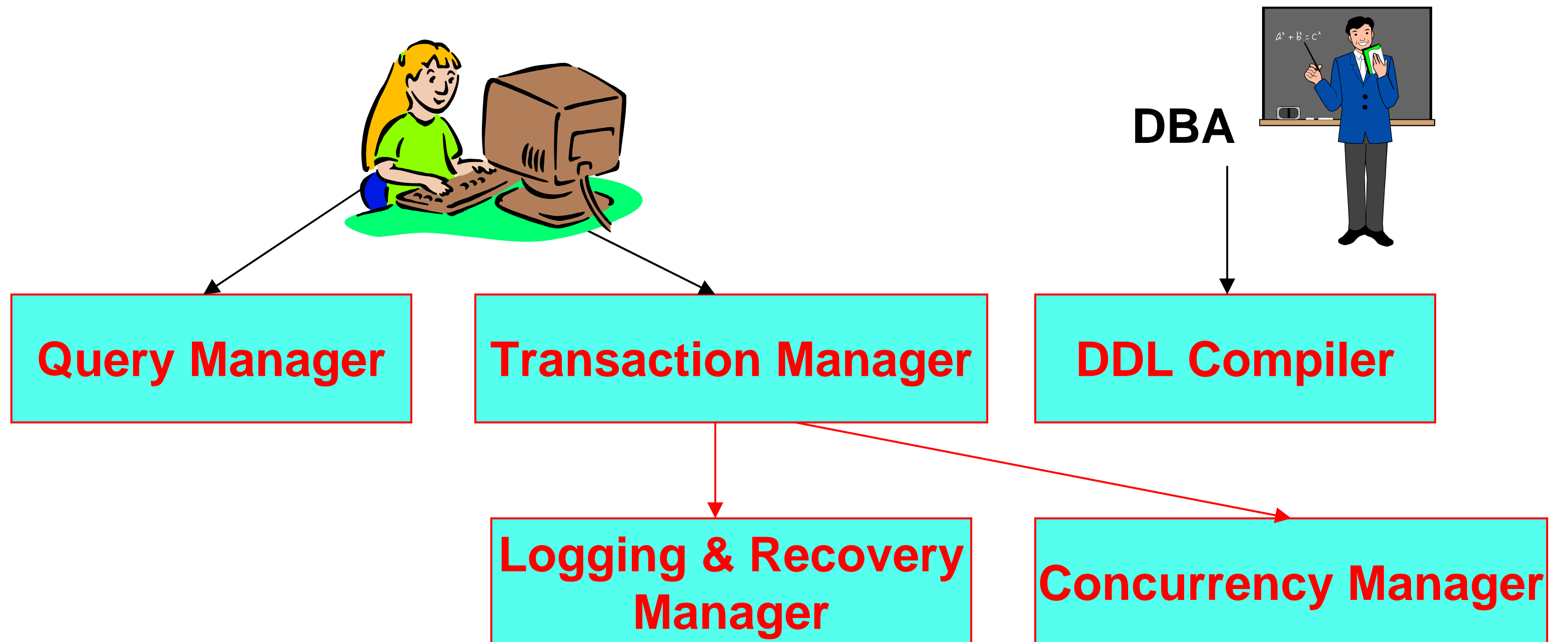
- In entrambi i casi tutte le operazioni elementari devono essere eseguite

Proprietà ACID di una transazione

- L'acronimo **ACID** indica le 4 proprietà che il DBMS deve garantire che valgano per ogni transazione:
  - Atomicity = una transazione è un'unità di elaborazione
    - Il DBMS garantisce che la transazione venga eseguita come un tutt'uno
  - Consistency = una transazione lascia il DB in uno stato consistente
    - Il DBMS garantisce che nessuno dei vincoli di integrità del DB venga violato
  - Isolation = una transazione esegue indipendentemente dalle altre
    - Se più transazioni eseguono in concorrenza, il DBMS garantisce che l'effetto netto è equivalente a quello di una qualche esecuzione sequenziale delle stesse
  - Durability = gli effetti di una transazione che ha terminato correttamente la sua esecuzione devono essere persistenti nel tempo
    - Il DBMS deve proteggere il DB a fronte di guasti

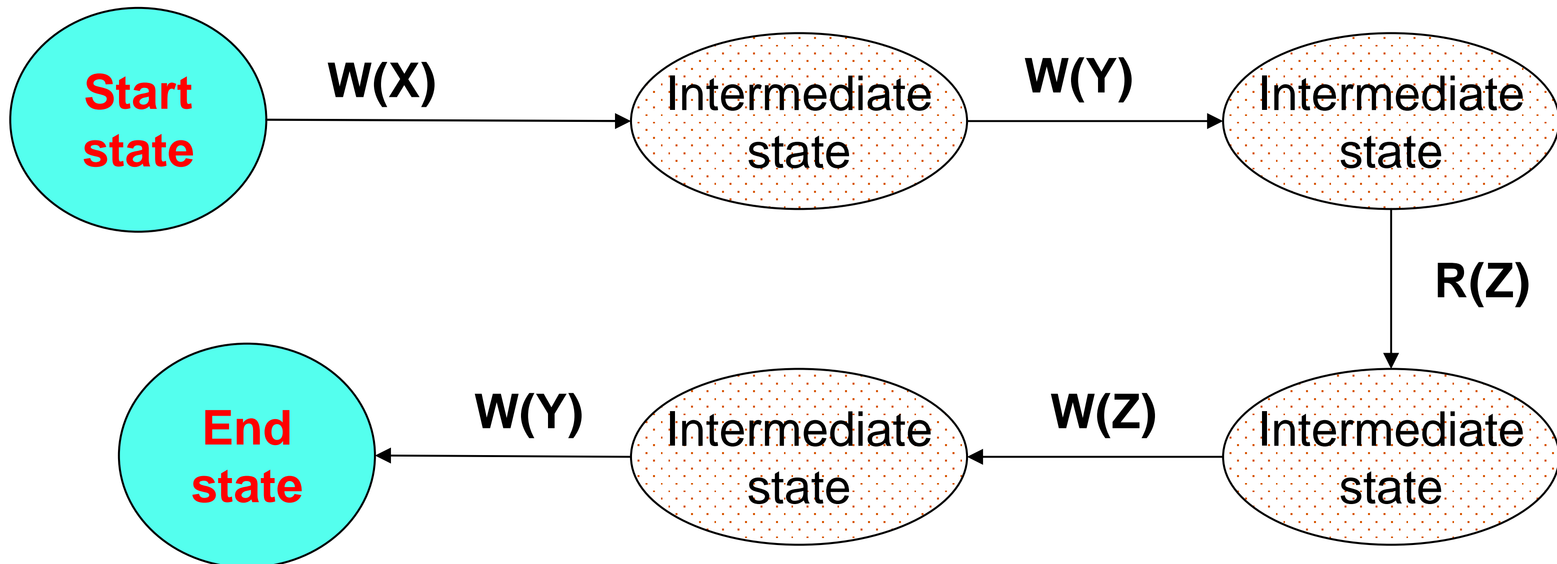
Proprietà ACID e moduli di un DBMS

Transaction Manager :	coordina l'esecuzione delle transazioni, ricevendo i comandi SQL ad esse relativi
Logging & Recovery Manager:	si fa carico di Atomicity e Durability
Concurrency Manager:	garantisce l'Isolation
DDL Compiler:	genera i controlli per la Consistency



Modello delle transazioni

- Nel modello che consideriamo una transazione viene vista come una sequenza di operazioni elementari di lettura (R) e scrittura (W) di oggetti (tuple) del DB che, a partire da uno stato iniziale consistente del DB, porta il DB in un nuovo stato finale consistente

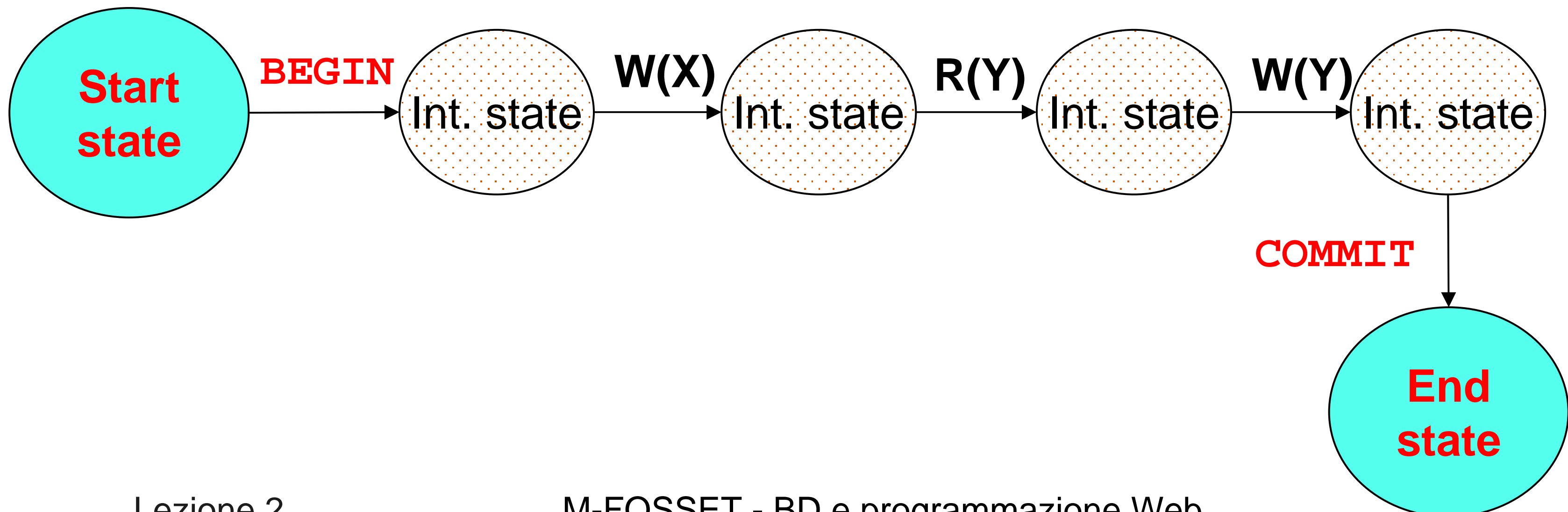


- In generale gli stati intermedi in cui si trova il DB non è richiesto che siano consistenti

Possibili esiti di una transazione (1)

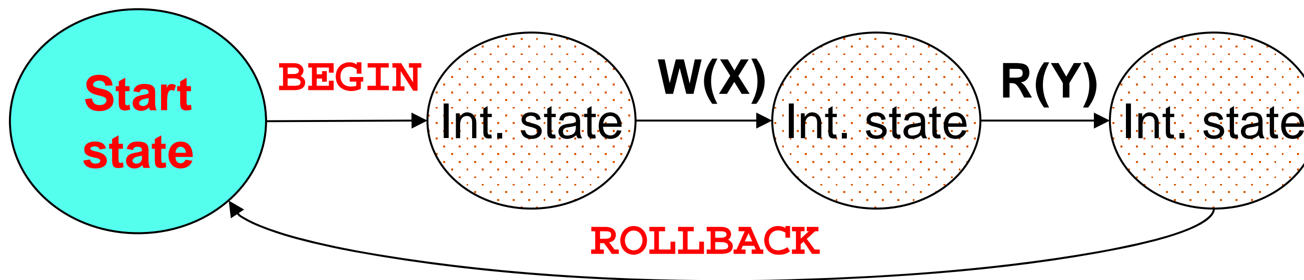
- Nel modello considerato una transazione (il cui inizio viene indicato da **`BEGIN [WORK]`**, o **`START TRANSACTION`**, o è implicito) può avere solo 2 esiti:
  - Terminare correttamente:

    Questo avviene solo quando l'applicazione, dopo aver eseguito tutte le proprie operazioni, esegue una particolare istruzione SQL, detta **`COMMIT`** (o **`COMMIT WORK`**), che comunica "ufficialmente" al Transaction Manager il termine delle operazioni



Possibili esiti di una transazione (2)

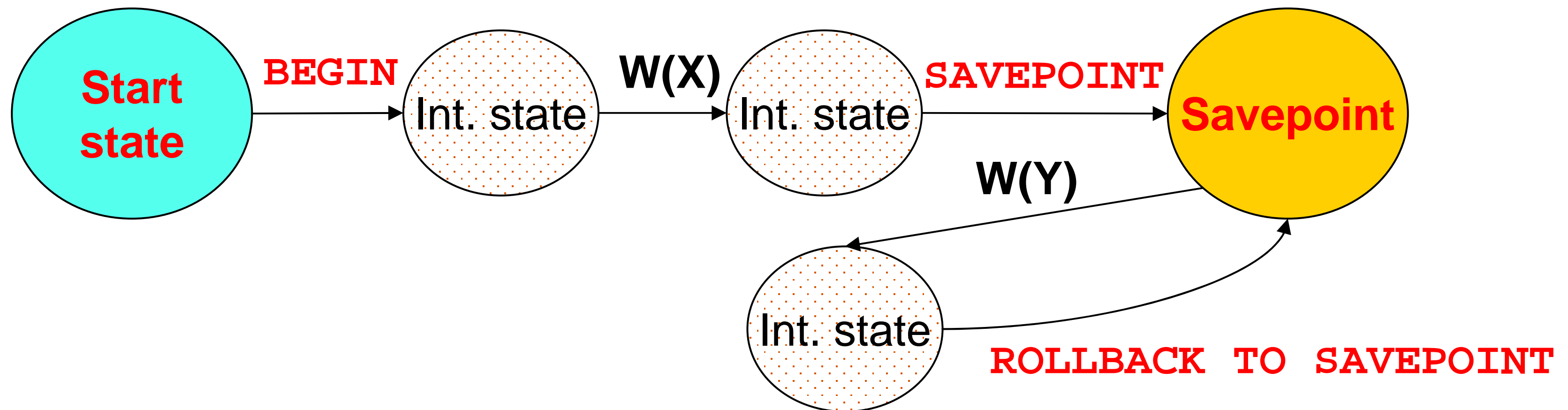
- Terminare non correttamente (anticipatamente); sono possibili 2 casi:
  - È la transazione che, per qualche motivo, decide che non ha senso continuare e quindi "abortisce" eseguendo l'istruzione SQL **`ROLLBACK`** (o **`ROLLBACK WORK`**)
  - È il sistema che non è in grado (ad es. per un guasto o per la violazione di un vincolo) di garantire la corretta prosecuzione della transazione, che viene quindi abortita



- Se per qualche motivo la transazione non può terminare correttamente la sua esecuzione il DBMS deve "disfare" (UNDO) le eventuali modifiche da essa apportate al DB

Transazioni con Savepoint

- Il modello di transazioni di SQL è in realtà essere più articolato; in particolare è definito il concetto di "savepoint", che permette di disfare solo parzialmente il lavoro svolto



Esempio di transazione con Savepoint

```
BEGIN WORK
SELECT * FROM Department
INSERT INTO Department(DeptNo,DeptName,AdmrDept)
                  VALUES ('X00',‘nuovo dept 1','A00')
SAVEPOINT pippo
SELECT * FROM Department -- qui include ‘X00’
INSERT INTO Department(DeptNo,DeptName,AdmrDept)
                  VALUES ('Y00','nuovo dept 2','A00')
SELECT * FROM Department -- qui anche ‘Y00’
ROLLBACK WORK TO SAVEPOINT pippo
SELECT * FROM Department -- qui solo ‘X00’
COMMIT WORK
```

Esecuzione seriale e concorrente

- Un DBMS, dovendo supportare l'esecuzione di diverse transazioni che accedono a dati condivisi, potrebbe eseguire tali transazioni in sequenza ("serial execution")
- In alternativa, il DBMS può eseguire più transazioni in concorrenza, alternando l'esecuzione di operazioni di una transazione con quella di operazioni di altre transazioni ("interleaved execution")
- Eseguire più transazioni concorrentemente è necessario per garantire buone prestazioni:
  - Ad es., mentre una transazione è in attesa del completamento di una operazione di I/O, un'altra può utilizzare la CPU, il che porta ad aumentare il "throughput" (n. transazioni elaborate nell'unità di tempo) del sistema
  - Banalizzando, se si ha una transazione "breve" e una "lunga", l'esecuzione concorrente porta a ridurre il tempo medio di risposta del sistema

Riduzione del tempo di risposta

- T1 è “lunga”, T2 è “breve”; per semplicità ogni riga della tabella è un’unità di tempo

T2 richiede a time = 2 di iniziare

time	T1	T2
1	R(X1)	
2	W(X1)	
…		
999	R(X500)	
1000	W(X500)	
1001	Commit	
1002		R(Y)
1003		W(Y)
1004		Commit

Tempo medio di risposta = (1001 + (1004-1))/2 = 1002

time	T1	T2
1	R(X1)	
2		R(Y)
3		W(Y)
4		Commit
5	W(X1)	
…		
1002	R(X500)	
1003	W(X500)	
1004	Commit	

Tempo medio di risposta = (1004 + 3)/2 = 503.5

Isolation: gestire la concorrenza

- Quando più transazioni vengono eseguite in concorrenza, si ha "isolation" se non vi è interferenza tra le stesse, ovvero se gli effetti sul DB sono equivalenti a quelli che si avrebbero con una esecuzione serializzata
- Nel caso di transazioni che interferiscono tra loro si possono avere 4 tipi base di problemi (dal più grave al meno grave):

- Lost Update: un aggiornamento viene perso
- Dirty Read: viene letto un dato che "non esiste" nel DB
  (si legge un dato scritto da una transazione ancora in esecuzione)
- Unrepeatable Read: letture successive di uno stesso dato sono tra loro inconsistenti
- Phantom Row: vengono selezionati dei dati di interesse, ma tra questi ne manca qualcuno (*phantom*)
  ("query non ripetibile": la stessa query restituisce risultati diversi se eseguita più volte all'interno della stessa transazione)

Lost Update

- Il seguente schedule mostra un caso tipico di lost update, in cui per comodità si evidenziano anche le operazioni che modificano il valore del dato X e si mostra come varia il valore di X nel DB

T1	X	T2
R(X)	1	
X=X-1	1	
	1	R(X)
	1	X=X-1
W(X)	0	
Commit	0	
	0	W(X)
	0	Commit

Questo update viene perso!

- Problema: T2 legge il valore di X prima che T1 (che lo ha già letto) lo modifichi

Dirty Read

- In questo caso il problema è che una transazione legge un dato “che non c’è”:

T1	X	T2
R(X)	0	
X=X+1	0	
W(X)	1	
	1	R(X)
Rollback	0	
	0	…
	0	Commit

Questa lettura è “sporca”!

- Quanto svolto da T2 si basa su un valore di X “intermedio”, e quindi non stabile
- Le conseguenze sono impredicibili (dipende cosa fa T2) e si presenterebbero anche se T1 non abortisse

Unrepeatable Read

- Ora il problema è che una transazione legge due volte un dato e trova valori diversi:

Le 2 letture sono tra loro inconsistenti!

T1	X	T2
R(X)	0	
	0	R(X)
	1	X=X+1
	1	W(X)
	1	Commit
R(X)	1	
Commit	1	

- Anche in questo caso si possono avere gravi conseguenze
- Lo stesso problema si presenta per transazioni di “analisi”
  - Ad esempio T1 somma l’importo di 2 conti correnti mentre T2 esegue un trasferimento di fondi dall’uno all’altro (T1 potrebbe quindi riportare un totale errato)

Phantom Row

- Questo caso si può presentare quando vengono inserite o cancellate tuple che un'altra transazione dovrebbe logicamente considerare
  - Nell'esempio la tupla t4 è un "phantom", in quanto T1 "non la vede"

T1:

```
UPDATE Prog
SET    Sede = 'Firenze'
WHERE  Sede = 'Bologna'
```

T2:

```
INSERT INTO Prog
VALUES ('P03','Bologna')
```

Prog

CodProg	Citta	
P01	Milano	t1
P01	Bologna	t2
P02	Bologna	t3
P03	Bologna	t4

T1 "non vede" questa tupla!

T1	T2
R(t2)	
R(t3)	
…	
W(t2)	
W(t3)	
	Insert(t4)
…	
Commit	
	Commit

Livelli di isolamento in SQL

- Scegliere di operare a un livello di isolamento in cui si possono presentare dei problemi ha il vantaggio di aumentare il grado di concorrenza raggiungibile, e quindi di migliorare le prestazioni
- Lo standard SQL definisce 4 livelli di isolamento (YES significa che il problema può presentarsi):

Isolation Level	Lost Update	Dirty Read	Unrepeatable Read	Phantom
`SERIALIZABLE`	NO	NO	NO	NO
`REPEATABLE READ`	NO	NO	NO	**YES**
`READ COMMITTED`	NO	NO	**YES**	**YES**
`READ UNCOMMITTED`	NO	**YES**	**YES**	**YES**

Transazioni in MySQL

- Di default ogni istruzione SQL è una transazione a sé
- Per modificare tale comportamento:

  `SET AUTOCOMMIT = 0`

  oppure iniziando esplicitamente una transazione (**`START TRANSACTION`**)

- Il livello di default è **`REPEATABLE READ`**; per cambiarlo ad es. a **`SERIALIZABLE`**, si usa l'istruzione SQL

  `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`

Transazioni in PostgreSQL

- **SET AUTOCOMMIT TO OFF** oppure iniziando esplicitamente una transazione (**START TRANSACTION**)
- In PostgreSQL è possibile richiedere ognuno dei 4 livelli (il default è **READ COMMITTED**), ma ne vengono supportati solo 2, come da tabella:

Richiesto	Effettivo
SERIALIZABLE	SERIALIZABLE
REPEATABLE READ	SERIALIZABLE
READ COMMITTED	READ COMMITTED
READ UNCOMMITTED	READ COMMITTED

- Il motivo è dovuto alla modalità con cui PostgreSQL gestisce transazioni in concorrenza, noto come Multiversion Concurrency Control (MVCC)

- Lezione 3: aspetti "fisici" relativi alla gestione delle transazioni (e ovviamente altro…)