Il linguaggio SQL: viste e tabelle derivate

Sistemi Informativi T

Versione elettronica: 04.5.SQL.viste.pdf

DB di riferimento 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

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 valutata dinamicamente ogni volta che si fa riferimento alla vista

```
CREATE VIEW ProgSedi(CodProg,CodSede)
AS      SELECT P.CodProg,S.Sede
        FROM   Prog P, Sedi S
        WHERE  P.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 si ricordano i seguenti:
  - 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

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

- Per evitare di ripetere i dati anagrafici, si decide di modificare lo schema del DB sostituendo alla tabella **EsamiSIT** le due seguenti:

  StudentiSIT(Matr,Cognome,Nome)
  ProveSIT(Matr,DataProva,Voto)

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

```
CREATE VIEW EsamiSIT(Matr,Cognome,Nome,DataProva,Voto)
AS      SELECT S.*,P.DataProva,P.Voto
        FROM   StudentiSIT S, ProveSIT 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(*))

Query complesse che usano VIEW (3)

- Con le viste è inoltre possibile risolvere query che richiedono "piu' passi di raggruppamento", ad es:

  Per ogni valore (arrotondato) di stipendio medio, numero delle sedi che pagano tale stipendio

- Occorre aggregare **prima** per sede, **poi** per valore di stipendio medio

```
CREATE VIEW StipSedi(Sede,AvgStip)
AS      SELECT   Sede, AVG(Stipendio)
        FROM     Imp
        GROUP BY Sede

SELECT AvgStip, COUNT(*) AS NumSedi
FROM   StipSedi
GROUP BY AvgStip
```

StipSedi

Sede	AvgStip
S01	1275
S02	1733
S03	1000

AvgStip	NumSedi
1275	1
1733	1
1000	1

Aggiornamento di viste

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

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

NumImp

Sede	NImp
S01	4
S02	3
S03	1

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

- Cosa significa? Non si può fare!

Aggiornabilità di viste (1)

- Una vista è di fatto una funzione che calcola un risultato y a partire da un'istanza di database **r**, y = V(**r**)
- L'aggiornamento di una vista, che trasforma y in y', può essere eseguito solo se è univocamente definita la nuova istanza **r'** tale che y' = V(**r'**), e questo corrisponde a dire che la vista è "invertibile", ossia **r'** = $V^{-1}(y')$
- Data la complessità del problema, di fatto ogni DBMS pone dei limiti 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)

- La precisazione che è il blocco più esterno della query di definizione che non deve contenere, ad es., dei join ha importanti conseguenze. Ad esempio, la 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 (1)

- 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

Viste con CHECK OPTION (2)

```
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')
WITH CHECK OPTION
```

- In questo modo l'inserimento

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

viene impedito

Tipi di CHECK OPTION (1)

- Si presenta spesso il caso in cui una vista V1 sia definita in termini di un'altra vista V2, magari a sua volta definita in termini di altre viste, ecc.
- Se si crea V1 specificando la clausola **WITH CHECK OPTION**, il DBMS verifica che la nuova tupla t inserita soddisfi sia la definizione di V1 che quella di V2 (e di tutte le altre eventuali viste da cui V1 dipende), indipendentemente dal fatto che V2 sia stata a sua volta definita **WITH CHECK OPTION**
- Questo comportamento di default è equivalente a definire V1

WITH CASCADED CHECK OPTION

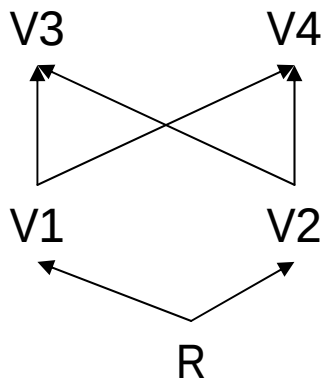
- Lo si può alterare definendo V1

WITH LOCAL CHECK OPTION

- Ora 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**

Tipi di CHECK OPTION (2)

- Si consideri il seguente grafo di dipendenze tra views e tables:



Vista	Definita come…
V1	WITH CHECK OPTION
V2	-
V3	WITH LOCAL CHECK OPTION
V4	WITH CHECK OPTION

- La tabella riassume le definizioni di quali viste vengono verificate quando si inserisce una tupla in una data vista

INSERT INTO	Si controllano…
V1	V1
V2	-
V3	V1,V3
V4	V1,V2,V4

Table expressions (1)

- 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"

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)
                FROM     Imp
               GROUP BY Sede) AS SM(Sede,MaxStip)
WHERE    I.Sede = SM.Sede
  AND    I.Stipendio = SM.MaxStip
GROUP BY SM.Sede,SM.MaxStip
```

SM

Sede	MaxStip
S01	2000
S02	2500
S03	1000

Table expressions (2)

- Anche le table expressions possono essere usate per query che richiedono diversi passi di aggregazione

  Per ogni valore (arrotondato) di stipendio medio, numero delle sedi che pagano tale stipendio

```
SELECT AvgStip, COUNT(*) AS NumSedi
FROM  (SELECT    Sede, AVG(Stipendio)
       FROM     Imp
       GROUP BY Sede) AS StipSedi(Sede,AvgStip)
GROUP BY AvgStip
```

Table expressions correlate (1)

- Una table expression può essere correlata a un'altra tabella che la precede nella clausola FROM
  - In DB2 è necessario utilizzare la parola riservata TABLE o LATERAL

Per ogni sede, la somma degli stipendi pagati agli analisti

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

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

- 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)
```

- Nel caso generale la clausola WITH supporta la definizione di più c.t.e.:

```
WITH CTE1(...) AS (...), CTE2(...) AS (...) ...
```

WITH e interrogazioni ricorsive (1)

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

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

- La query è ricorsiva e pertanto non è esprimibile in algebra relazionale, in quanto richiede un numero di (self-)join non noto a priori
- La formulazione mediante common table expressions definisce la vista temporanea (ricorsiva) **`Antenati(Persona,Avo)`** facendo l'unione di:
  - una "subquery base" non ricorsiva (che inizializza **`Antenati`** con le tuple di **`Genitori`**)
  - una "subquery ricorsiva" che 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

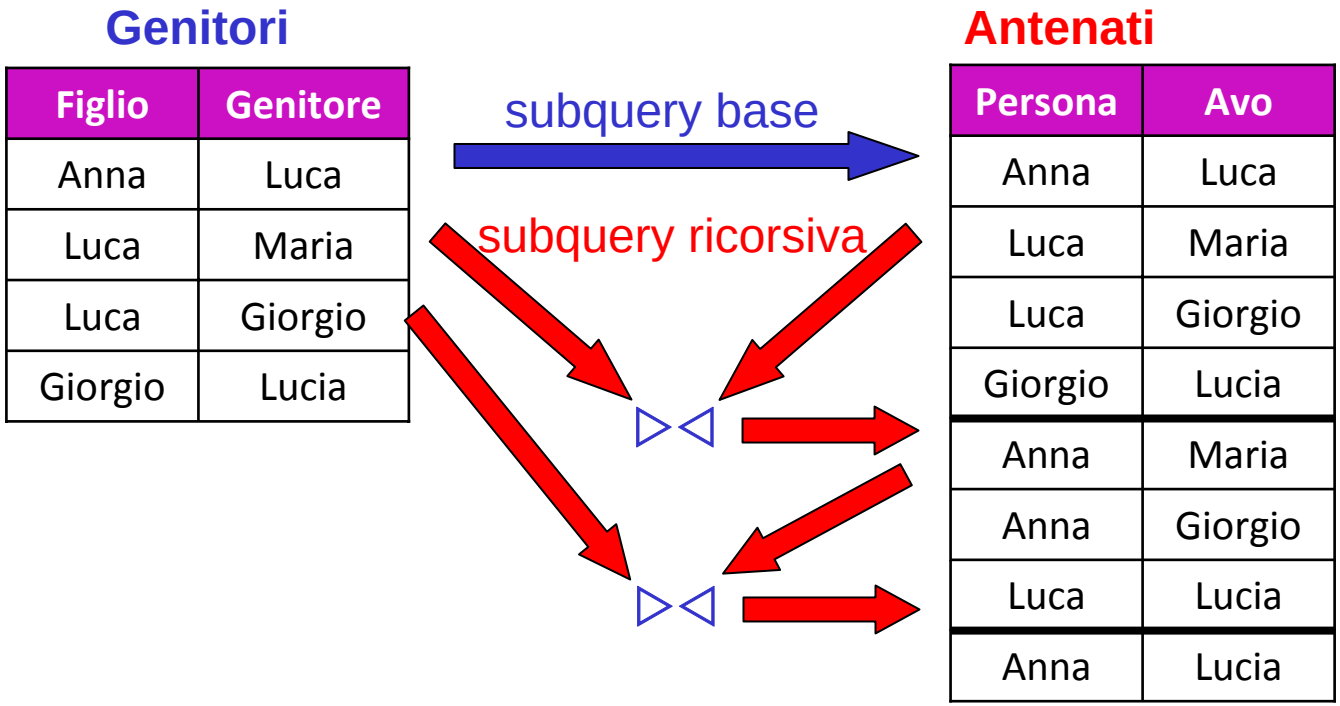
WITH e interrogazioni ricorsive (2)

```
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**



Tipi notevoli di interrogazioni ricorsive

- Il caso precedentemente visto corrisponde a un DB "aciclico" in cui prima o poi l'esecuzione è garantita terminare
- Inoltre non viene calcolata nessuna informazione aggiuntiva per i "percorsi" trovati
  - **Percorso**: intuitivamente è la sequenza di tuple di cui si fa il join e che generano una tupla nel risultato finale (nell'esempio visto corrisponde a una "linea genealogica" specifica)

- Vediamo nel seguito due casi notevoli
  - Query ricorsive con informazione sui percorsi (lunghezza, "costo", ecc.)
  - Condizione di stop per query ricorsive su DB "ciclici"

Informazione sui percorsi (1)

- Il caso più semplice prevede l'aggiunta di un informazione sulla lunghezza (distanza, livello, ecc.) del percorso
- Nel caso già visto l'output sarebbe:

Genitori

Figlio	Genitore
Anna	Luca
Luca	Maria
Luca	Giorgio
Giorgio	Lucia

Antenati

Persona	Avo	Lungh
Anna	Luca	1
Luca	Maria	1
Luca	Giorgio	1
Giorgio	Lucia	1
Anna	Maria	2
Anna	Giorgio	2
Luca	Lucia	2
Anna	Lucia	3

Informazione sui percorsi (2)

- La lunghezza parte da 1 e si incrementa di 1 a ogni passo, quindi:

```
WITH Antenati(Persona,Avo,Lungh)
AS   ((SELECT Figlio,Genitore,1 -- caso base: Lungh = 1
       FROM   Genitori)
      UNION ALL
       (SELECT G.Figlio,A.Avo,A.Lungh+1
        FROM   Genitori G, Antenati A
        WHERE  G.Genitore = A.Persona))
SELECT *
FROM   Antenati
WHERE  Persona = 'Anna'
```

Persona	Avo	Lungh
Anna	Luca	1
Anna	Maria	2
Anna	Giorgio	2
Anna	Lucia	3

Informazione sui percorsi (3)

- In alcuni casi l'informazione sui percorsi è un "costo", oppure qualche altra grandezza cumulativa
- Esempio: nella relazione PARTI, per ogni parte si dice quante unità di altre parti servono per costruirla/assemblarla
- SI vuole sapere quante unità di una parte servono **complessivamente** per costruire un'altra parte
  - Nell'esempio: 22 = 2*5 + 3*4

Parti

Parte	Subparte	Qta
P1	P2	2
P1	P3	3
P2	P4	5
P3	P4	4

Composto	Componente	QtaTot
P1	P2	2
P1	P3	3
P2	P4	5
P3	P4	4
P1	**P4**	**22**

Informazione sui percorsi (4)

- In questi casi la soluzione prevede:
  - l'uso della ricorsione per calcolare l'informazione di ogni singolo percorso
    - P1-P2-P4: 2*5=10; P1-P3-P4: 3*4=12
  - l'aggregazione delle informazione dei singoli percorsi nella query che usa la vista ricorsiva

Parti

Parte	Subparte	Qta
P1	P2	2
P1	P3	3
P2	P4	5
P3	P4	4

Percorsi

Composto	Componente	Qty
P1	P2	2
P1	P3	3
P2	P4	5
P3	P4	4
P1	P4	10
P1	P4	12

Informazione sui percorsi (5)

- Ad ogni passo si moltiplica per quello che si aggiunge al percorso:

```
WITH Percorsi(Composto,Componente,Qty)
AS   ((SELECT Parte,Subparte,Qta
       FROM   Parti)
      UNION ALL
       (SELECT H.Composto,P.Subparte,H.Qty*P.Qta
        FROM   Parti P, Percorsi H
        WHERE  H.Componente = P.Parte))
--
SELECT Composto,Componente,SUM(Qty) AS QtaTot
FROM   Percorsi
GROUP BY Composto,Componente
```

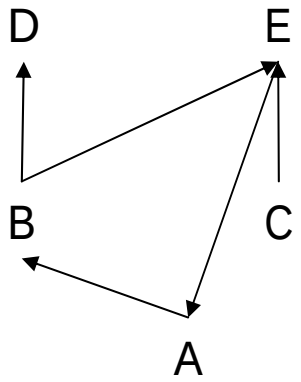
Database ciclici

- Nel caso di DB ciclici occorre prestare particolare attenzione
- E' infatti necessario prevedere una condizione di stop, altrimenti il rischio è la non terminazione!
- Non facendo uso di altri strumenti non presentati nel corso, occorre inserire nella subquery ricorsiva un predicato che, prima o poi, sia falso per tutte le nuove tuple che si andrebbero ad aggiungere alla vista

- Casi tipici:
  - limitazione sulla lunghezza massima dei percorsi
  - limitazione sul "costo"

Limitazione sulla lunghezza dei percorsi (1)

- Il grafo in figura contiene un ciclo (percorso di lunghezza infinita):
  - A-B-E-A-B-E-A-…
- Fissando un limite sulla lunghezza dei percorsi si generano solo percorsi di lunghezza finita, quindi l'esecuzione prima o poi termina
- Per non perdere risultati, il limite va scelto in modo da non escludere percorsi "interessanti"



Siti

From	To
A	B
B	D
B	E
C	E
E	A

Limitazione sulla lunghezza dei percorsi (2)

- Posto il limite ad es. a 3 si ha:

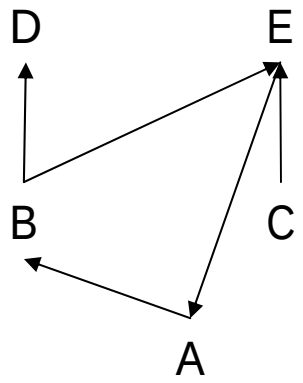
```
WITH Percorsi(Da,A,Lungh)
AS   ((SELECT From,To,1
       FROM   Siti)
      UNION ALL
       (SELECT P.Da,S.To,P.Lungh+1
        FROM   Siti S, Percorsi P
        WHERE  P.A = S.From
        AND P.Da <> S.To
        AND P.Lungh+1 <= 3))
SELECT *
FROM   Percorsi
```

Siti

From	To
A	B
B	D
B	E
C	E
E	A

Percorsi

Da	A	Lungh
A	B	1
B	D	1
B	E	1
C	E	1
E	A	1
A	D	2
A	E	2
B	A	2
C	A	2
E	B	2
E	D	3

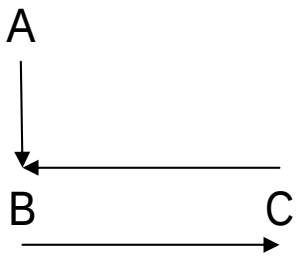


Limitazione sulla lunghezza dei percorsi (3)

- La condizione **`P.Da <> S.To`** serve ad evitare di inserire informazione inutile nel risultato
- Benché in alcuni casi particolari possa essere usata anche come condizione di terminazione, nel caso generale non funziona!

Siti

From	To
A	B
B	C
C	B



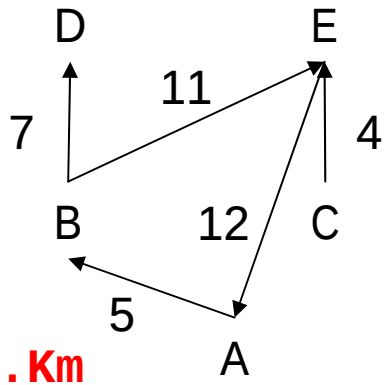
Percorsi

Da	A	Lungh
A	B	1
B	C	1
C	B	1
A	C	2
A	B	3
A	C	4
A	B	5
…	…	…

Limitazione sul “costo”

- In maniera simile si ragiona se si vuole imporre un limite al costo dei percorsi

```
WITH Percorsi(Da,A,TotKm)
AS   ((SELECT From,To,Km
       FROM   Paesi)
      UNION ALL
       (SELECT P.Da,S.To,P.TotKm+S.Km
        FROM   Paesi S, Percorsi P
        WHERE  P.A = S.From
        AND P.Da <> S.To
        AND P.TotKm+S.Km <= 17))
SELECT *
FROM   Percorsi
```



Paesi

From	To	Km
A	B	5
B	D	7
B	E	11
C	E	4
E	A	10

Percorsi

Da	A	Lungh
…	…	…
A	D	12
A	E	16
C	A	16
E	B	17

- In generale ci possono essere più percorsi tra 2 siti e va considerato quello a costo minore…

Riassumiamo:

- Le viste sono tabelle virtuali, interrogabili come le altre, ma soggette a limiti per ciò che riguarda gli aggiornamenti
- Una table expression è una subquery che definisce una tabella derivata utilizzabile nella clausola FROM
- Una common table expression è una "vista temporanea" che può essere usata in una query come se fosse a tutti gli effetti una VIEW
- Mediante common table expression è anche possibile formulare interrogazioni ricorsive, definendo "viste temporanee ricorsive" come unione del risultato di una "subquery base" e una "subquery ricorsiva"