Laboratorio 7

SQL – Ricorsione e Trigger

rif. paragrafi 7.5 e 10.2 libro

Corso di Basi di Dati

A.A. 2012/2013

Presented by
Matteo Lissandrini
matteo.lissandrini@gmail.com

Courtesy of:
Francesco Corcoglioniti
corcoglioniti@disi.unitn.it

Ripasso –WITH e Ricorsione

- Clausola WITH:

```
WITH [RECURSIVE] relazione1 (attr1, ..., attrN) AS ... SELECT
query ...,
     [RECURSIVE] relazione2 (...) AS ...
SELECT ...
```

- WITH permette di definire relazioni temporanee usabili nella SELECT
- Le relazioni RECURSIVE possono referenziare direttamente o indirettamente se stesse nella SELECT che le definisce
  - –concettualmente, tali query sono valutate iterativamente
  - –ogni iterazione può aggiungere nuove tuple alle relazioni ricorsive
  - –il processo termina dopo un'iterazione che non aggiunge nuove tuple
- Limitazioni:
  - –**linear recursion** – nella definizione di una relazione ricorsiva si può citare soltanto una relazione mutuamente ricorsiva (e.g. la relazione stessa o altra relazione che causa loop nel *dependency graph* – vedere libro)
  - –**monotonicità** – le definizioni devono essere tali che ciascuna iterazione possa solo aggiungere tuple e mai togliere tuple aggiunte in precedenza

Ripasso –Trigger

- Sintassi:

```
CREATE TRIGGER name
AFTER|BEFORE|INSTEAD OF event
REFERENCING
   OLD ROW|TABLE AS nome1
   NEW ROW|TABLE AS nome2
FOR EACH ROW|STATEMENT
WHEN (...condizione a livello di tupla...)
... codice SQL (tra BEGIN e END se più
comandi) ...
```

- Eventi:
  - **INSERT ON** tabella
  - **DELETE ON** tabella
  - **UPDATE OF** [(attr1, ...,attrM)] **ON** tabella

Ripasso – Funzioni SQL utili

- Utente corrente:
  - **`CURRENT_USER`** – ritorna ad esempio "postgres"
- Funzioni su data e ora
  - **`CURRENT_DATE`** – ritorna ad esempio "2011-10-18"
  - **`CURRENT_TIME`** – ritorna ad esempio "22:15:22.624+02"
  - **`CURRENT_TIMESTAMP`** – ritorna ad esempio "2011-10-18 22:15:10.103+02"
  - **`EXTRACT`**`(part` **`FROM`** `time_exp)` – estra una componente temporale; part può essere YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, TIMEZONE_MINUTE
- Funzioni su stringhe
  - `stringa1` **`||`** `stringa2` – concatenzione
  - **`CHAR_LENGTH`**`(stringa)` – ritorna lunghezza stringa
  - **`POSITION`**`(substring` **`IN`** `string)` – ritorna la posizione di substring all'interno di string partendo da 1, se contenuta, altrimenti 0
  - **`SUBSTRING`**`(string` **`FROM`** `start [`**`FOR`** `length])` – estrae una sottostringa a partire dal carattere in posizione start, opz. con al più length caratteri
  - **`LOWER`**`(string)` – trasforma string in lowercase
  - **`UPPER`**`(string)` – trasforma string in uppercase
  - **`TRIM`**`([`**`LEADING`**`|`**`TRAILING`**`|`**`BOTH`**`]` **`FROM`** `string)` – rimuove gli spazi a inizio/fine stringa (default: entrambi gli estremi)

Note sugli Esercizi Proposti

- PostgreSQL supporta una forma ristretta di ricorsione:
  - nella WITH, le tabelle RECURSIVE devono essere definite secondo la forma (SELECT caso_base) UNION (SELECT passo_ricorsivo), dove la seconda SELECT non può contenere aggregati e può citare la tabella ricorsiva solo una volta
  - gli esercizi proposti sono comunque tutti eseguibili in PostgreSQL

- PostgreSQL supporta solo una forma ristretta di trigger
  - no REFERENCING …, no evento UPDATE su attributi specifici, azione deve essere chiamata a funzione

- Esercizi 1, 2, 3
  - gli esercizi vertono sulla ricorsione e si appoggiano allo schema 'recursion'
  - usare lo script create_recursion_db.sql
    - http://j.mp/BasiDati07

- Esercizi 4, 5
  - eseguibili solo su carta, non in PostgreSQL

Esercizio 1 – Città

Sia data la tabella: tratta (citta1, citta2, distanza)

La tabella contiene la distanza stradale in Km tra coppie di città principali italiane (indicate con la sigla della corrispondente provincia, in uppercase); per ogni coppia sono riportate due tuple da citta1 a citta2 e viceversa, con la stessa distanza.

Usando la ricorsione in SQL, si esprimano le seguenti query:

1. Elencare tutte le città raggiungibili da Roma (citta = 'RM'), anche indirettamente
2. Elencare tutte le città raggiungibili da Roma (anche indirettamente) e distanti meno di 500 Km
3. Elencare tutte le città raggiungibili da Roma (anche indirettamente) e distanti meno di 500 Km, riportando distanza minima e percorso (suggerimento: usare un campo stringa con l'elenco delle città attraversate)
4. Elencare tutte le città raggiungibili da Roma in max 3 tratte, riportando per ciascuna il numero di tratte percorse
5. Costruire una tabella delle distanze chilometriche, riportando per ogni coppia di città raggiungibili tra loro la relativa distanza minima (opzionale: riportare anche il percorso corrispondente; suggerimento: evitare i loop!)
6. Elencare tutti i percorsi con relative distanze per andare da Aosta (AO) a Reggio Calabria (RC), evitando di passare per la stessa città intermedia più volte

Esercizio 1 – Soluzioni (1/5)

1. Elencare tutte le città raggiungibili da Roma (citta = 'RM'), anche indirettamente

```
WITH RECURSIVE raggiungibile (citta) AS
     ( ( SELECT CAST('RM' AS CHAR(2)) )
       UNION
       ( SELECT citta2
         FROM   raggiungibile r JOIN tratta t ON r.citta = t.citta1 ))

SELECT citta
FROM   raggiungibile;
```

2. Elencare tutte le città raggiungibili da Roma (anche ind.) distanti meno di 500 Km

```
WITH RECURSIVE raggiungibile (citta, distanza) AS
     ( ( SELECT citta2, distanza
         FROM   tratta
         WHERE  citta1 = 'RM' AND distanza < 500 )
       UNION
       ( SELECT t.citta2, (r.distanza + t.distanza) AS distanza
         FROM   raggiungibile r JOIN tratta t ON r.citta = t.citta1
         WHERE  r.distanza + t.distanza < 500 ) )

SELECT DISTINCT citta
FROM   raggiungibile
```

Esercizio 1 – Soluzioni (2/5)

3. Elencare tutte le città raggiungibili da Roma (anche indirettamente) e distanti meno di 500 Km, riportando distanza minima e percorso (suggerimento: usare un campo stringa con l'elenco delle città attraversate)

```
WITH RECURSIVE raggiungibile (citta, distanza, percorso) AS
     ( ( SELECT citta2, distanza, ('RM ' || citta2) AS percorso
         FROM   tratta
         WHERE  citta1 = 'RM' AND distanza < 500 )
       UNION
       ( SELECT t.citta2, (r.distanza + t.distanza) AS distanza,
                (r.percorso || ' ' || t.citta2) AS percorso
         FROM   raggiungibile r JOIN tratta t ON r.citta = t.citta1
         WHERE  r.distanza + t.distanza < 500 ) )

SELECT r1.citta, r1.distanza, r1.percorso
FROM   raggiungibile r1
WHERE  r1.distanza = ( SELECT MIN(r2.distanza)
                       FROM   raggiungibile r2
                       WHERE  r2.citta = r1.citta );
```

Esercizio 1 – Soluzioni (3/5)

4. Elencare tutte le città raggiungibili da Roma in max 3 tratte, riportando per ciascuna il numero di tratte percorse

```
WITH RECURSIVE raggiungibile (citta, num_tratte) AS
     ( ( SELECT citta2, 1
         FROM   tratta
         WHERE  citta1 = 'RM' )
       UNION
       ( SELECT t.citta2, (r.num_tratte + 1) AS num_tratte
         FROM   raggiungibile r JOIN tratta t ON r.citta = t.citta1
         WHERE  num_tratte <= 2) )

SELECT   citta, MIN(num_tratte) AS min_num_tratte
FROM     raggiungibile
GROUP BY citta
```

Esercizio 1 – Soluzioni (4/5)

5. Costruire una tabella delle distanze chilometriche, riportando per ogni coppia di città raggiungibili tra loro la relativa distanza minima (opzionale: riportare anche il percorso corrispondente; suggerimento: evitare i loop!)

```
WITH RECURSIVE percorso (partenza, arrivo, distanza, tappe) AS
     ( ( SELECT citta1, citta2, distanza,
                (citta1 || ' ' || citta2) AS tappe
         FROM   tratta )
       UNION
       ( SELECT p.partenza, t.citta2,
                (p.distanza + t.distanza) AS distanza,
                (p.tappe || ' ' || t.citta2) AS tappe
         FROM   percorso p JOIN tratta t ON p.arrivo = t.citta1
         WHERE  POSITION(t.citta2 IN p.tappe) = 0 ) )
SELECT   *
FROM     percorso p1
WHERE    NOT EXISTS ( SELECT *
                      FROM   percorso p2
                      WHERE  p1.partenza = p2.partenza AND
                             p1.arrivo = p2.arrivo AND
                             p1.distanza > p2.distanza )
ORDER BY p1.partenza, p1.arrivo
```

Esercizio 1 – Soluzioni (5/5)

6. Elencare tutti i percorsi con relative distanze per andare da Aosta (AO) a Reggio Calabria (RC), evitando di passare per la stessa città intermedia più volte

```
WITH RECURSIVE raggiungibile (citta, distanza, tappe) AS
     ( ( SELECT citta2, distanza, ('AO ' || citta2) AS tappe
         FROM   tratta
         WHERE  citta1 = 'AO' )
       UNION
       ( SELECT t.citta2, (r.distanza + t.distanza) AS distanza,
                (r.tappe || ' ' || t.citta2) AS tappe
         FROM   raggiungibile r JOIN tratta t ON r.citta = t.citta1
         WHERE  POSITION(t.citta2 IN r.tappe) = 0 ) )

SELECT   distanza, tappe
FROM     raggiungibile
WHERE    citta = 'RC'
ORDER BY distanza
```

Esercizio 2 – Gerarchia

Sia data la tabella: dipendente(matricola, nome, manager, stipendio)

La tabella contiene l'elenco dei dipendenti di un'azienda (sia semplici impiegati che manager); il campo manager fa riferimento alla matricola del manager responsabile di un certo dipendente.

Usando la ricorsione in SQL, si esprimano le seguenti query:

1. Elencare tutti i manager a cui afferisce, direttamente o indirettamente, il dipendente 'Filippo Verdi '
2. Elencare eventuali manager che percepiscono uno stipendio inferiore di un dipendente a loro afferente
3. Elencare per ogni dipendente quanti dipendenti sono da esso amministrati (direttamente o indirettamente) e qual'è l'importo complessivo speso per i relativi stipendi
4. Elencare numero dipendenti, stipendio minimo, medio e massimo e totale stipendi per livello gerarchico, partendo dal boss il quale non è associato a nessun responsabile

Esercizio 2 – Soluzioni (1/4)

1. Elencare tutti i manager a cui afferisce, direttamente o indirettamente, il dipendente 'Filippo Verdi ‘

```
WITH RECURSIVE afferenza(manager) AS
     ( ( SELECT manager
         FROM   dipendente
         WHERE  nome = 'Filippo Verdi' )
       UNION
       ( SELECT d.manager
         FROM   afferenza a JOIN dipendente d
                            ON a.manager = d.matricola
         WHERE  d.manager IS NOT NULL ) )

SELECT nome
FROM   afferenza a JOIN dipendente d ON a.manager = d.matricola
```

Esercizio 2 – Soluzioni (2/4)

2. Elencare eventuali manager che percepiscono uno stipendio inferiore di un dipendente a loro afferente

```
WITH RECURSIVE afferenza(matricola, manager) AS
     ( ( SELECT matricola, manager
         FROM   dipendente )
       UNION
       ( SELECT a.matricola, d.manager
         FROM   afferenza a JOIN dipendente d
                            ON a.manager = d.matricola
         WHERE  d.manager IS NOT NULL ) )

SELECT d2.nome, d2.stipendio, d1.nome, d1.stipendio
FROM   afferenza a JOIN dipendente d1 on a.matricola = d1.matricola
                   JOIN dipendente d2 on a.manager = d2.matricola
WHERE  d1.stipendio > d2.stipendio
```

Esercizio 2 – Soluzioni (3/4)

3. Elencare per ogni dipendente quanti dipendenti sono da esso amministrati (direttamente o indirettamente) e qual'è l'importo complessivo speso per i relativi stipendi

```
WITH RECURSIVE afferenza(matricola, manager) AS
     ( ( SELECT matricola, manager
         FROM   dipendente )
       UNION
       ( SELECT a.matricola, d.manager
         FROM   afferenza a JOIN dipendente d
                            ON a.manager = d.matricola
         WHERE  d.manager IS NOT NULL ) )

SELECT   m.nome, COUNT(*) num_dipendenti,
         SUM(d.stipendio) AS tot_stipendi
FROM     afferenza a JOIN dipendente m ON a.manager = m.matricola
                     JOIN dipendente d ON a.matricola = d.matricola
GROUP BY m.nome
```

Esercizio 2 – Soluzioni (4/4)

4. Elencare numero dipendenti, stipendio minimo, medio e massimo e totale stipendi per livello gerarchico, partendo dal boss il quale non è associato a nessun responsabile

```
WITH RECURSIVE gerarchia (livello, matricola, stipendio) AS
     ( ( SELECT 1 AS livello, matricola, stipendio
         FROM   dipendente
         WHERE  manager IS NULL )
       UNION
       ( SELECT (g.livello + 1) AS livello, d.matricola, d.stipendio
         FROM   gerarchia g JOIN dipendente d
                            ON d.manager = g.matricola ) )

SELECT   livello, COUNT(*) AS num_dipendenti,
         MIN(stipendio) AS min_stipendio,
         MAX(stipendio) AS max_stipendio,
         AVG(stipendio) AS avg_stipendio,
         SUM(stipendio) AS tot_stipendi
FROM     gerarchia
GROUP BY livello
ORDER BY livello
```

Esercizio 3 – Numeri

Sfruttando la ricorsione e tenendo presente che, in SQL, la query 'SELECT costante AS attributo' (e.g. SELECT 1 AS numero) è una query valida che ritorna una tabella con una sola tupla di un solo attributo, si risolvano i seguenti quesiti:

1. Scrivere una query ricorsiva che elenchi tutti i numeri interi da 1 a 100
2. Sfruttando la stessa tecnica di cui sopra, scrivere una query che ritorni tutti i numeri primi minori di 100 (nota: in SQL l'espressione A % B ritorna il resto della divisione di A per B)
3. Si ipotizzi di avere una tabella vuota: studente(<u>matricola</u>, nome, cognome, anno), dove anno è l'anno di corso (1-5). Si scriva una INSERT con query SELECT ricorsiva per popolare la tabella con dati casuali. Si usi RANDOM() per generare un numero casuale tra 0 e 1 e ROUND() per arrotondare all'intero successivo.
4. Si ipotizzi di avere una tabella: fatturato (<u>anno</u>, importo). La tabella dovrebbe contenere i fatturati di un'impresa dal 1990 al 2010, tuttavia per alcuni anni mancano i corrispondenti fatturati. Si scriva una query SELECT che elenchi gli anni per cui non è stato precisato un fatturato (suggerimento: usare lo stesso procedimento adottato per generare i numeri interi per enumerare gli anni dal 1990 al 2010)

Esercizio 3 – Soluzioni (1/4)

1. Scrivere una query ricorsiva che elenchi tutti i numeri interi da 1 a 100

```
WITH RECURSIVE number(num) AS
      ( ( SELECT 1 )
        UNION
        ( SELECT (num + 1) AS num
          FROM   number
          WHERE  num < 100 ) )

SELECT * FROM number;
```

Esercizio 3 – Soluzioni (2/4)

2. Sfruttando la stessa tecnica di cui sopra, scrivere una query che ritorni tutti i numeri primi minori di 100

```
WITH RECURSIVE number(num) AS
      ( ( SELECT 1 )
          UNION
        ( SELECT (num + 1) AS num
          FROM   number
          WHERE  num < 100 ) )

SELECT n1.num
FROM   number n1
WHERE  NOT EXISTS ( SELECT *
                    FROM   number n2
                    WHERE  n2.num > 1 AND n2.num < n1.num AND
                           n1.num % n2.num = 0 )
```

Esercizio 3 – Soluzioni (3/4)

3. Si ipotizzi di avere una tabella vuota: impiegato (matricola, nome, cognome, anno). Si scriva una INSERT con query SELECT ricorsiva per popolare la tabella con dati casuali. Si usi RANDOM() per generare un numero casuale tra 0 e 1 e ROUND() per arrotondare all'intero successivo.

```
INSERT INTO studente (matricola, nome, cognome, anno)
WITH RECURSIVE number(num) AS
     ( ( SELECT 1 )
       UNION
       ( SELECT (num + 1) AS num
         FROM   number
         WHERE  num < 100 ) )

SELECT num AS matricola,
       ('nome' || round(random() * 100)) AS nome,
       ('cognome' || round(random() * 100)) AS cognome,
       (1 + round(random() * 4.49)) AS anno
FROM   number;
```

Esercizio 3 – Soluzioni (4/4)

4. Si ipotizzi di avere una tabella: fatturato (anno, importo). La tabella dovrebbe contenere i fatturati di un'impresa dal 1990 al 2010, tuttavia per alcuni anni mancano i corrispondenti fatturati. Si scriva una query SELECT che elenchi gli anni per cui non è stato precisato un fatturato

```
WITH RECURSIVE calendario(anno) AS
     ( ( SELECT 1990 )
       UNION
       ( SELECT (anno + 1) AS anno
         FROM   calendario
         WHERE  anno < 2010 ) )

SELECT anno
FROM   calendario NATURAL LEFT JOIN fatturato
WHERE  importo IS NULL
```

Esercizio 4 – Prodotti

rif. esercizi 7.5.1 (quesiti b-e)

Sia dato lo schema:

product (<u>model</u>, maker, type)
pc (<u>model</u>, speed, ram, hd, price)
laptop (<u>model</u>, speed, ram, hd, screen, price)
printer (<u>model</u>, color, type, price)

Si scrivano le istruzioni SQL per definire i seguenti trigger, il cui compito è vietare l'operazione che ha fatto scattare il trigger:

1. In concomitanza con l'inserimento di una stampante, verificare che non ci sia una stampante con prezzo più basso dello stesso tipo (inkjet/laser, colori/bn)
2. In concomitanza con l'inserimento di un PC, verificare che non ci sia già una tupla con lo stesso modello in laptop e printer
3. In concomitanza con operazioni di modifica alla tabella PC, imporre che il prezzo medio dei PC per ciascun produttore sia sempre superiore a 500$
4. In concomitanza con la modifica della dimensione di HD di un PC, verificare che la nuova dimensione sia grande almeno 100 volte la dimensione della RAM

Esercizio 4 – Soluzioni (1/4)

1. In concomitanza con l'inserimento di una stampante, verificare che non ci sia una stampante con prezzo più basso dello stesso tipo (inkjet/laser, colori/bn)

```
CREATE TRIGGER insert_printers_with_lower_price
  BEFORE INSERT ON printer
  REFERENCING NEW ROW AS n
  FOR EACH ROW
WHEN (EXISTS ( SELECT *
               FROM   printer p
               WHERE  p.price < n.price AND
                      p.type = n.type AND p.color = n.color ) )
ROLLBACK;
```

Osservazioni. *La scelta tra BEFORE e AFTER è qui indifferente; si noti tuttavia che, se il vincolo fosse stato su prezzo minore o uguale, allora la scelta di BEFORE sarebbe obbligata, poichè usando AFTER la nuova tupla inserita sarebbe presente anch'essa in 'printer' e sarebbe selezionata dalla query. Si sceglie FOR EACH ROW per comodità: così facendo è sufficiente cercare una stampante con prezzo inferiore al prezzo nuovo (che è uno solo). Con FOR EACH STATEMENT, si tratterebbe invece di fare un JOIN tra la NEW table e printer, cercando coppie di stampanti (new, old) di stesso tipo ma con prezzo della old inferiore.*

Esercizio 4 – Soluzioni (2/4)

2. In concomitanza con l'inserimento di un PC, verificare che non ci sia già una tupla con lo stesso modello in laptop e printer

```
CREATE TRIGGER pc_model_only_in_pc_table
  BEFORE INSERT ON pc
  REFERENCING NEW ROW AS p
  FOR EACH ROW
WHEN ( EXISTS     ( SELECT *
                    FROM   laptop l
                    WHERE  n.model = l.model )
       OR EXISTS  ( SELECT *
                    FROM   printer s
                    WHERE  n.model = s.model ) )
ROLLBACK;
```

Osservazioni. *La scelta tra BEFORE e AFTER è indifferente in questo caso; si preferisce BEFORE in modo da risparmiare un pò di lavoro nel caso la tupla non vada inserita, in quanto il trigger scatta così prima dell'inserimento vero e proprio e non bisogna in tal caso invertire gli effetti di una INSERT con la ROLLBACK. La scelta FOR EACH ROW è qui più comoda, in quanto permette di considerare e ricercare nelle altre tabelle un solo model (quello del nuovo PC)*

Esercizio 4 – Soluzioni (3/4)

3. In concomitanza con operazioni di modifica alla tabella PC, imporre che il prezzo medio dei PC per ciascun produttore sia sempre superiore a 500$

```
CREATE TRIGGER avg_pc_price_greater_than_500
  AFTER INSERT ON pc
  FOR EACH STATEMENT
WHEN ( EXISTS ( SELECT   maker
                FROM     product NATURAL JOIN pc
                GROUP BY maker
                HAVING   AVG(price) <= 500 ) )
ROLLBACK;
```

Altri due trigger vanno definiti con la specifica:

```
AFTER UPDATE OF price ON pc
AFTER DELETE ON pc
```

Osservazioni. *In questo caso ne la condizione ne l'azione del trigger dipendono dalla particolare modifica (e.g. la particolare tupla inserita, cancellata o modificata). Essi dipendono soltanto dal nuovo stato del DB. Per questo motivo, risulta comodo operare con un trigger statement-level e – poichè interessa il nuovo stato – si usa la specifica AFTER.*

Esercizio 4 – Soluzioni (4/4)

4. In concomitanza con la modifica della dimensione di HD di un PC, verificare che la nuova dimensione sia grande almeno 100 volte la dimensione della RAM

```
CREATE TRIGGER new_hd_size_1000_times_ram_size
  BEFORE UPDATE OF hd ON pc
  REFERENCING NEW ROW AS newpc
  FOR EACH ROW
WHEN (newpc.hd < 1000 * newpc.ram)
ROLLBACK;
```

Osservazioni. *La scelta tra BEFORE e AFTER è indifferente; si preferisce BEFORE per evitare un pò di lavoro nel caso la tupla non debba essere modificata. Si sceglie FOR EACH ROW perchè risulta più comodo testare tupla per tupla invece che congiuntamente su tutte le tuple modificate.*

Esercizio 5 – Navi da battaglia

rif. esercizio 7.5.3 (quesiti a-e)

Sia dato lo schema:

classes (class, type, country, num_guns, bore, displacement)
ships (name, class, launched)
outcomes (ship, battle, result)
battles (name, date)

Si scrivano le istruzioni SQL per definire i seguenti trigger, il cui compito è vietare l'operazione che ha fatto scattare il trigger:

1. Quando si inserisce una nuova classe con stazza (displacement) maggiore di 20000 t, modificare la stazza riportandola a 20000 t
2. In concomitanza con l'inserimento di una nuova classe, inserire anche una nave con lo stesso nome e data di varo NULL in ships
3. In concomitanza a inserimenti o modifiche a ships, impedire che un paese abbia più di 30 navi
4. In concomitanza con l'inserimento di una tupla in outcomes per cui non ci sono corrispondenti in ships e/o battles, inserire automaticamente delle tuple in queste relazioni usando NULL ove necessario
5. Vietare che una nave risulti partecipante ad una battaglia successiva alla battaglia in cui è stata affondata

Esercizio 5 – Soluzioni (1/5)

1. Quando si inserisce una nuova classe con stazza (displacement) maggiore di 20000 t, modificare la stazza riportandola a 20000 t

```
CREATE TRIGGER max_displacement_20000_after_insert
  AFTER INSERT ON classes
  REFERENCING NEW ROW AS c
  FOR EACH ROW
  WHEN (c.displacement > 20000)
UPDATE classes
SET    displacement = 20000
WHERE  class = c.class;
```

Un altro trigger `max_displacement_20000_after_update` va definito con la specifica

```
AFTER UPDATE OF displacement ON classes
```

Osservazioni. *Si sceglie AFTER in modo da poter agire a valle dell'inserimento della tupla modificando la stazza se necessario. Si sceglie un row-level trigger per comodità e perchè così facendo si può scrivere facilmente la condizione WHEN (in caso di statement-level trigger, si dovrebbe usare WHEN(TRUE) o query più complessa che verifichi se almeno una tupla inserita ha stazza da cambiare).*

Esercizio 5 – Soluzioni (2/5)

2. In concomitanza con l'inserimento di una nuova classe, inserire anche una nave con lo stesso nome e data di varo NULL in ships

```
CREATE TRIGGER homonym_ship_for_each_class
  AFTER INSERT ON classes
  REFERENCING NEW TABLE AS c
  FOR EACH STATEMENT
  WHEN (TRUE)
INSERT INTO ships(name, class)
SELECT class, class
FROM   c
```

Osservazioni. *Nell'ipotesi che ci sia un vincolo di integrità referenziale da ships a classes, occorre lavorare con AFTER in modo tale che, quando si inserisce la nave omonima, esista già la classe relativa nel DB. Se tale vincolo non esiste (come nel DB distribuito in aula), allora non c'è differenza tra AFTER e BEFORE. Non essendoci una condizione da valutare, si preferisce usare uno statement-level trigger in quanto è in genere più performante fare una singola attivazione di trigger ed una singola insert piuttosto che tante attivazioni e insert singole.*

Esercizio 5 – Soluzioni (3/5)

3. In concomitanza a inserimenti o modifiche a ships, impedire che un paese abbia più di 30 navi

```
CREATE TRIGGER max_30_ships_after_insert_on_ships
  AFTER INSERT ON ships
  FOR EACH STATEMENT
WHEN ( EXISTS ( SELECT   country
                FROM     classes NATURAL JOIN ships
                GROUP BY country
                HAVING   COUNT(*) > 30 ) )
ROLLBACK;
```

Un altro trigger `trigger max_30_ships_after_update` va definito con la specifica `AFTER UPDATE OF class ON ship`

Osservazioni. *Si tratta di valutare una condizione sullo stato del DB dopo una modifica (cosa che si potrebbe fare con un'asserzione). Pertanto, si usa AFTER per poter operare sul nuovo stato del DB e si lavora a livello di STATEMENT in quanto è sufficiente un solo controllo a seguito dell'intera operazione di modifica (e non tupla per tupla).*

Esercizio 5 – Soluzioni (4/5)

4. In concomitanza con l'inserimento di una tupla in outcomes per cui non ci sono corrispondenti in ships e/o battles, inserire automaticamente delle tuple in queste relazioni usando NULL ove necessario

```
CREATE TRIGGER insert_missing_ship
  AFTER INSERT ON outcomes
  REFERENCING NEW ROW AS o
  FOR EACH ROW
WHEN ( o.ship <> NULL AND NOT EXISTS ( SELECT *
                                       FROM   ships
                                       WHERE  name = o.ship ) )
INSERT INTO ships (name) VALUES (o.ship);

CREATE TRIGGER insert_missing_battle
  AFTER INSERT ON outcomes
  REFERENCING NEW ROW AS o
  FOR EACH ROW
WHEN ( o.battle <> NULL AND NOT EXISTS ( SELECT *
                                         FROM   battles
                                         WHERE  name = o.battle ) )
INSERT INTO battles (name) VALUES (o.battle);
```

Esercizio 5 – Soluzioni (5/5)

5. Vietare che una nave risulti partecipante ad una battaglia successiva alla battaglia in cui è stata affondata

```
CREATE TRIGGER no_more_battles_after_sinking
  AFTER INSERT ON outcomes
  FOR EACH STATEMENT
WHEN ( EXISTS
       ( SELECT *
         FROM   outcomes o1 JOIN battles b1 ON o1.battle = b1.name,
                outcomes o2 JOIN battles b2 ON o2.battle = b2.name
         WHERE  o1.ship = o2.ship AND
                o1.result = 'sunk' AND b2.date > b1.date ) )
ROLLBACK;
```

Occorre definire altri due trigger con le specifiche:

```
AFTER UPDATE ON outcomes
AFTER UPDATE OF date ON battles
```

Osservazioni. *Si tratta di valutare una condizione sullo stato del DB a valle di certe modifiche su outcomes e battles. Si opera quindi con AFTER e a livello di statement.*

Esercizi dal libro

10.2.1	ricorsione, db Voli, riportato in slide seguenti
10.2.3	ricorsione, db Movies, riportato in slide seguenti
7.5.2	coperto totalmente da esercizio 4
7.5.2	coperto totalmente da esercizio 5
7.5.4	database Movies, riportato in slide seguenti

Esercizi dal libro – 10.2.1

Exercise 10.2.1: The relation

```
Flights(airline, frm, to, departs, arrives)
```

from Example 10.8 has arrival- and departure-time information that we did not consider. Suppose we are interested not only in whether it is possible to reach one city from another, but whether the journey has reasonable connections. That is, when using more than one flight, each flight must arrive at least an hour before the next flight departs. You may assume that no journey takes place over more than one day, so it is not necessary to worry about arrival close to midnight followed by a departure early in the morning.

a) Write the recursion in SQL.

~~b) Write this recursion in Datalog.~~

Esercizi dal libro – 10.2.3

Exercise 10.2.3: Suppose we have a relation

```
SequelOf(movie, sequel)
```

that gives the immediate sequels of a movie, of which there can be more than one. We want to define a recursive relation `FollowOn` whose pairs (x, y) are movies such that y was either a sequel of x, a sequel of a sequel, or so on.

a) Write the definition of `FollowOn` as a SQL recursion.

~~b) Write the definition of `FollowOn` as recursive Datalog rules.~~

c) Write a recursive SQL query that returns the set of pairs (x, y) such that movie y is a follow-on to movie x, but is not a sequel of x.

d) Write a recursive SQL query that returns the set of pairs (x, y) meaning that y is a follow-on of x, but is neither a sequel nor a sequel of a sequel.

! e) Write a recursive SQL query that returns the set of pairs (x, y) such that movie y is a follow-on of x but y has at most one follow-on.

! f) Write a recursive SQL query that returns the set of movies x that have at least two follow-ons. Note that both could be sequels, rather than one being a sequel and the other a sequel of a sequel.

Esercizi dal libro – 7.5.4

! **Exercise 7.5.4:** Write the following as triggers. In each case, disallow or undo the modification if it does not satisfy the stated constraint. The problems are based on our running movie example:

```
Movies(title, year, length, genre, studioName, producerC#)
StarsIn(movieTitle, movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)
```

You may assume that the desired condition holds before any change to the database is attempted. Also, prefer to modify the database, even if it means inserting tuples with `NULL` or default values, rather than rejecting the attempted modification.

a) Assure that the average length of all movies made in any year is no more than 150.

b) Assure that at all times, any star appearing in `StarsIn` also appears in `MovieStar`.

c) Assure that every movie has at least one male and one female star.

d) Assure that at all times every movie executive appears as either a studio president, a producer of a movie, or both.

e) Assure that the number of movies made by any studio in any year is no more than 50.