Triggers – Esercitazione 1

Nel seguente documento vengono mostrati alcuni esempi di trigger e di funzioni pgplSQL. Si ricorda che i trigger vengono eseguiti al verificarsi di certe condizioni definite dal progettista del database e che ogni volta che un trigger va in esecuzione viene invocata una particolare funzione definita in un linguaggio plSQL. In particolare, noi ci concentreremo sulle funzioni scritte nel linguaggio pgplSQL.

Archiviazione Barche

Il primo esempio che vedremo e' quello di un trigger che va in esecuzione ogni volta che viene cancellata una tupla dalla tabella `barche` e aggiunge la tupla eliminata alla tabella `ex_barche`. Innanzitutto e' necessario creare la tabella `ex_barche` con lo stesso schema della relazione della tabella `barche`.

Prima di tutto definiamo la funzione che il trigger richiamerà durante la sua esecuzione.

```
CREATE OR REPLACE FUNCTION archivia_barca() RETURNS trigger AS $archivia_barca$
    BEGIN
       -- Aggiorna la tabella delle barche non più in uso
       INSERT INTO ex_barche SELECT OLD.bid, OLD.bnome, OLD.colore;
       RETURN OLD;
    END
$archivia_barca$ LANGUAGE plpgsql;
```

Una volta creata la funzione possiamo definire il trigger come riportato di seguito.

```
CREATE TRIGGER archivia_barca
AFTER INSERT OR UPDATE OR DELETE ON barche
   FOR EACH ROW EXECUTE PROCEDURE archivia_barca();
```

Esercizio:

Creare una tabella `ex_velisti`, con lo stesso schema della relazione della tabella `velisti` e definire una funzione e un trigger che inseriscono nella tabella `ex_velisti` ogni tupla che viene cancellata dalla tabella `velisti`.

Calcolo Statistiche di Accesso al Database

Questo esempio mostra come mantenere delle statistiche di accesso al database ogni volta che viene effettuata un comando di `update`, di `insert` o di `delete` su una qualsiasi tabella del database. Prima di tutto e' necessario creare una tabella che conterrà le statistiche di accesso al database.

Lo schema della relazione `statistiche` e' riportato di seguito.

Column	*Type*
tipo_accesso	character(1)
nome_utente	character varying(20)
tempo	timestamp with time zone

Una volta creato lo schema della relazione `statistiche` si può procedere alla definizione della funzione che verra' invocata ogni volta che un trigger andrà in esecuzione. La funzione per l'aggiornamento delle statistiche e' riportata di seguito.

```
CREATE OR REPLACE FUNCTION statistiche_vela() RETURNS TRIGGER AS $statistiche_vela$
    BEGIN
       -- Crea una riga nella tabella che riflette le modifiche fatte
       -- nel database
       IF (TG_OP = 'DELETE') THEN
           INSERT INTO statistiche SELECT 'D', user, now();
           RETURN OLD;
       ELSIF (TG_OP = 'UPDATE') THEN
           INSERT INTO statistiche SELECT 'U', user, now();
           RETURN NEW;
       ELSIF (TG_OP = 'INSERT') THEN
           INSERT INTO statistiche SELECT 'I', user, now();
           RETURN NEW;
       END IF;
       RETURN NULL; -- Ignora il risultato perchè siamo dopo del trigger
END;
$statistiche_vela$ LANGUAGE plpgsql;
```

La variabile `TG_OP` presente nello script e' una variabile che viene settata automaticamente dal trigger che invoca questa funzione e contiene al suo interno l'operazione che ha scatenato il trigger stesso. In questo modo possiamo scrivere una sola funzione per gestire sia gli eventi dovuti ad un comando di `update`, ad un comando di `insert` e ad un comando di `delete`. La variabile `user` contiene invece il nome utente di chi ha eseguito il comando che ha poi scatenato il trigger. La funzione `now()` ritorna il tempo corrente nel formato `timestamp`.

Una volta definita la funzione di aggiornamento delle statistiche e' necessario definire un trigger che richiama questa funzione per ciascuna tabella di cui vogliamo monitorare le statistiche. Di seguito sono riportate le definizioni dei trigger per tutte le tabelle.

```
CREATE TRIGGER aggiorna_statistiche
AFTER INSERT OR UPDATE OR DELETE ON prenotazioni
   FOR EACH ROW EXECUTE PROCEDURE statistiche_vela();

CREATE TRIGGER aggiorna_statistiche
AFTER INSERT OR UPDATE OR DELETE ON barche
   FOR EACH ROW EXECUTE PROCEDURE statistiche_vela();

CREATE TRIGGER aggiorna_statistiche
AFTER INSERT OR UPDATE OR DELETE ON velisti
   FOR EACH ROW EXECUTE PROCEDURE statistiche_vela();

CREATE TRIGGER aggiorna_statistiche
AFTER INSERT OR UPDATE OR DELETE ON ex_barche
   FOR EACH ROW EXECUTE PROCEDURE statistiche_vela();

CREATE TRIGGER aggiorna_statistiche
AFTER INSERT OR UPDATE OR DELETE ON ex_velisti
   FOR EACH ROW EXECUTE PROCEDURE statistiche_vela();
```

E' interessante notare che se sia i triggers di aggiornamento delle statistiche che quelli di archiviazione delle tabelle `velisti` e `barche` sono definiti nel database, si può osservare il fenomeno dei triggers richiamati a cascata. Ad esempio, quando viene cancellata una tupla dalla tabella `barche`, vengono invocati sia il trigger di aggiornamento delle statistiche, sia il trigger per l'archiviazione del valore cancellato nella tabella `ex_barche`. L'inserimento nella tabella `ex_barche` della tupla cancellata da `barche`, scatena a sua volta un nuovo trigger di aggiornamento statistiche per la tabella `ex_barche`.
Nota: secondo la definizione del linguaggio pgplSQL se due trigger vengono invocati contemporaneamente, essi vengono eseguiti secondo l'ordine alfabetico dei loro nomi.

La variabile `TG_OP` presente nello script e' una variabile che viene settata automaticamente dal trigger che invoca questa funzione e contiene al suo interno l'operazione che ha scatenato il trigger stesso. In questo modo possiamo scrivere una sola funzione per gestire sia gli eventi dovuti ad un comando di `update`, ad un comando di `insert` e ad un comando di `delete`. La variabile `user` contiene invece il nome utente di chi ha eseguito il comando che ha poi scatenato il trigger. La funzione `now()` ritorna il tempo corrente nel formato `timestamp`.

Una volta definita la funzione di aggiornamento dellIn questo caso la successione temporale con cui vengono invocati i triggers quando si cancella una tupla dalla tabella `barche` e' la seguente:

1. TRIGGER aggiorna_statistiche (invocato dalla tabella Barche);
2. TRIGGER archivia_barca (invocato dalla tabella Barche);
3. TRIGGER aggiorna_statistiche (invocato dalla tabella Statistiche);

In particolare il trigger 3 e' invocato in cascata dal trigger 1.

Triggers – Esercitazione 2

Si considerino le relazioni prodotti, ordini e prodotti_correlati e i seguenti schemi delle relazioni:

prodotti (pid:*integer*, nome:*char[20]*, quantita:*integer*);

ordini (oid:*integer*, pid:*integer*, quantita:*integer*);

prodotti_correlati(pid:*integer*, cid:*integer*);

La relazione prodotti contiene l'id del prodotto, il suo nome e la quantità di quel prodotto attualmente presente in magazzino.

La relazione prodotti_correlati contiene l'id di due prodotti che sono tra loro correlati. Una tupla della relazione prodotti_correlati si può esprimere dicendo che il prodotto cid è correlato con il prodotto pid.

La relazione ordini contiene l'id unico dell'ordine, il pid del prodotto cui tale ordine si riferisce e la quantità da ordinare.

Si considerino come esempio le seguenti istanze legali delle relazioni prodotti e prodotti correlati:

prodotti		
pid	**nome**	**quantita**
1	Pizza	5
2	Coca Cola	4
3	Birra	3
4	Pasta	4
5	Sugo al Pesto	3
6	Sugo al Ragu	3

prodotti_correlati	
pid	**cid**
1	2
1	3
4	5
4	6

Esercizio: Definire i vincoli di foreign key in modo opportuno, sapendo che un ordine non può essere fatto se il prodotto non è presente nella tabella prodotti e che due prodotti per essere correlati devono entrambi essere presenti nella tabella prodotti.

Di seguito è riportato il codice di una funzione che emula il comportamento di un contatore auto-incrementante.

Esercizio: inserire la funzione di autocontatore come trigger invocato sull'evento INSERT nella tabella ordini.

```
 CREATE OR REPLACE FUNCTION contatore() RETURNS TRIGGER AS $contatore$
    DECLARE
       record ordini%ROWTYPE;
       max ordini.oid%TYPE;
    BEGIN
       max := 0;
       FOR record IN SELECT *
                     FROM ordini
                  LOOP
                      IF (ordini.oid > max) THEN
                           max := record.oid;
                      END IF;
                  END LOOP;
       NEW.oid := max + 1;

       RETURN NEW;
END;
$contatore$ LANGUAGE plpgsql;

CREATE TRIGGER aggiorna_oid
BEFORE INSERT ON ordini
   FOR EACH ROW EXECUTE PROCEDURE contatore();
```

Esercizio: Definire un trigger che esegue una funzione di aggiornamento ordini ogni volta che la quantità di un certo prodotto scende al disotto di 3 pezzi. La funzione di aggiornamento ordini è una funzione che inserisce un nuovo ordine nella tabella ordini. Notare che questo trigger non si dovrà preoccupare di settare in modo opportuno il numero progressivo che identifica l'id dell'ordine in quanto questo viene fatto dal trigger sopra descritto. E' sufficiente che il trigger che crea un nuovo ordine metta un valore qualsiasi nel campo oid.

Soluzione:

```
CREATE OR REPLACE FUNCTION crea_ordine() RETURNS TRIGGER AS $crea_ordine$
    DECLARE
      -- Nessuna dichiarazione
    BEGIN
       IF (NEW.quantita < 3) THEN
          INSERT INTO ordini SELECT '0', NEW.pid, '20';
       END IF;
       RETURN OLD;
END;
$crea_ordine$ LANGUAGE plpgsql;

CREATE TRIGGER aggiorna_ordini
AFTER INSERT OR UPDATE ON prodotti
   FOR EACH ROW EXECUTE PROCEDURE crea_ordine();
```

Esercizio: Definire un trigger che esegue una funzione di aggiornamento ordini ogni volta che la quantità di un certo prodotto scende al disotto di 3 pezzi e che fa un ordine relativo anche ad ogni prodotto che è correlato con il prodotto la cui quantità è scesa al disotto di 3.

Soluzione:

```
CREATE OR REPLACE FUNCTION crea_ordine() RETURNS TRIGGER AS $crea_ordine$
    DECLARE
      id prodotti_correlati.cid%TYPE;
    BEGIN
       IF (NEW.quantita < 3) THEN
          INSERT INTO ordini SELECT '0', NEW.pid, '20';
          FOR id IN ( SELECT cid
                        FROM prodotti_correlati
                        WHERE pid = NEW.pid )
              LOOP
                  INSERT INTO ordini SELECT '0', id, '20';
              END LOOP;
       END IF;
       RETURN OLD;
END;
$crea_ordine$ LANGUAGE plpgsql;

CREATE TRIGGER aggiorna_ordini
AFTER INSERT OR UPDATE ON prodotti
   FOR EACH ROW EXECUTE PROCEDURE crea_ordine();
```