Modification to Views Via Triggers

Oracle allows us to “intercept” a modification to a view through an instead-of trigger.

Example

\[
\begin{align*}
\text{Likes} & (\text{drinker, beer}) \\
\text{Sells} & (\text{bar, beer, price}) \\
\text{Frequents} & (\text{drinker, bar})
\end{align*}
\]

CREATE VIEW Synergy AS

\[
\begin{align*}
\text{SELECT} & \text{ Likes.drinker, Likes.beer,} \\
& \text{ Sells.bar} \\
\text{FROM} & \text{ Likes, Sells, Frequents} \\
\text{WHERE} & \text{ Likes.drinker =} \\
& \text{ Frequents.drinker AND} \\
& \text{ Likes.beer = Sells.beer AND} \\
& \text{ Sells.bar = Frequents.bar;}
\end{align*}
\]
CREATE TRIGGER ViewTrig
INSTEAD OF INSERT ON Synergy
FOR EACH ROW
BEGIN
  INSERT INTO Likes VALUES(:new.drinker, :new.beer);
  INSERT INTO Sells(bar, beer)
    VALUES(:new.bar, :new.beer);
  INSERT INTO Frequents VALUES(
    :new.drinker, :new.bar);
END;
.
run
SQL Triggers

- Read in text.
- Some differences, including:
  1. The Oracle restriction about not modifying the relation of the trigger or other relations linked to it by constraints is not present in SQL (but Oracle is real; SQL is paper).
  2. The action in SQL is a list of (restricted) SQL statements, not a PL/SQL statement.
PL/SQL

- Oracle’s version of PSM (Persistent, Stored Modules).
  - Use via sqlplus.
- A compromise between completely procedural programming and SQL’s very high-level, but limited statements.
- Allows local variables, loops, procedures, examination of relations one tuple at a time.
- Rough form:
  ```
  DECLARE  
    declarations
  BEGIN  
    executable statements
  END;
  .
  run;
  ```
- DECLARE portion is optional.
- Dot and run (or a slash in place of run;) are needed to end the statement and execute it.
Simplest Form: Sequence of Modifications

```
Likes(drinker, beer)

BEGIN
    INSERT INTO Likes
    VALUES('Sally', 'Bud');
    DELETE FROM Likes
    WHERE drinker = 'Fred' AND
        beer = 'Miller';

END;
```

run;
Procedures

Stored database objects that use a PL/SQL statement in their body.

Procedure Declarations

```
CREATE OR REPLACE PROCEDURE <name>(<arglist>) AS
  <declarations>
  BEGIN
    <PL/SQL statements>
  END;

.. run;
```
• Argument list has name-mode-type triples.
  ✦ Mode: IN, OUT, or IN OUT for read-only, write-only, read/write, respectively.
  ✦ Types: standard SQL + generic types like NUMBER = any integer or real type.
  ✦ Since types in procedures must match their types in the DB schema, you should generally use an expression of the form
    
    relation.attribute%TYPE
  
    to capture the type correctly.
Example

A procedure to take a beer and price and add it to Joe’s menu.

\[ \text{Sells(bar, beer, price)} \]

CREATE PROCEDURE joeMenu(
    b IN Sells.beer%TYPE,
    p IN Sells.price%TYPE
) AS
    BEGIN
        INSERT INTO Sells
            VALUES('Joe’s Bar', b, p);
    END;
.
run;

- Note “run” only stores the procedure; it doesn’t execute the procedure.
Invoking Procedures

A procedure call may appear in the body of a PL/SQL statement.

- Example:

  ```sql
  BEGIN
    joeMenu('Bud', 2.50);
    joeMenu('MooseDrool', 5.00);
  END;
  .
  run;
  ```
Assignment
Assign expressions to declared variables with $\texttt{:=}$. 

Branches

IF $<$condition$>$ THEN
    $<$statement(s)$>$
ELSE
    $<$statement(s)$>$
END IF;

- But in nests, use $\texttt{ELSIF}$ in place of $\texttt{ELSE IF}$.

Loops

LOOP
    . . .
    EXIT WHEN $<$condition$>$
    . . .
END LOOP;
Queries in PL/SQL

1. *Single-row selects* allow retrieval into a variable of the result of a query that is guaranteed to produce one tuple.

2. *Cursors* allow the retrieval of many tuples, with the cursor and a loop used to process each in turn.
Single-Row Select

- Select-from-where in PL/SQL must have an INTO clause listing variables into which a tuple can be placed.

- It is an error if the select-from-where returns more than one tuple; you should have used a cursor.

Example

Find the price Joe charges for Bud (and drop it on the floor).

\[
\text{Sells(bar, beer, price)}
\]

DECLARE

\[\text{p Sells.price\%TYPE;}\]
BEGIN

\[\text{SELECT price INTO p FROM Sells WHERE bar = 'Joe's Bar' AND beer = 'Bud';}\]

END;
.
run
Cursors

Declare by:

```
CURSOR <name> IS
    select-from-where statement
```

- Cursor gets each tuple from the relation produced by the select-from-where, in turn, using a *fetch statement* in a loop.

  ✦ Fetch statement:

  ```
  FETCH <cursor name> INTO
    variable list;
  ```

- Break the loop by a statement of the form:

  ```
  EXIT WHEN <cursor name> NOTFOUND;
  ```

  ✦ True when there are no more tuples to get.

- Open and close the cursor with *OPEN* and *CLOSE*. 
Example

A procedure that examines the menu for Joe’s Bar and raises by $1.00 all prices that are less than $3.00.

\[
\text{Sells}(\text{bar}, \text{beer}, \text{price})
\]

- This simple price-change algorithm can be implemented by a single \texttt{UPDATE} statement, but more complicated price changes could not.
CREATE PROCEDURE joeGouge() AS
    theBeer Sells.beer%TYPE;
    thePrice Sells.price%TYPE;
    CURSOR c IS
        SELECT beer, price
        FROM Sells
        WHERE bar = 'Joe''s bar';
    BEGIN
        OPEN c;
        LOOP
            FETCH c INTO theBeer, thePrice;
            EXIT WHEN c%NOTFOUND;
            IF thePrice < 3.00 THEN
                UPDATE Sells
                SET price = thePrice + 1.00
                WHERE bar = 'Joe''s Bar'
                AND beer = theBeer;
            END IF;
        END LOOP;
        CLOSE c;
    END;
    run
Row Types

Anything (e.g., cursors, table names) that has a tuple type can have its type captured with %ROWTYPE.

- We can create temporary variables that have tuple types and access their components with dot.
- Handy when we deal with tuples with many attributes.
Example

The same procedure with a tuple variable \( bp \).

```
CREATE PROCEDURE joeGouge() AS
    CURSOR c IS
        SELECT beer, price
        FROM Sells
        WHERE bar = 'Joe''s bar';
    bp c%ROWTYPE;
    BEGIN
        OPEN c;
        LOOP
            FETCH c INTO bp;
            EXIT WHEN c%NOTFOUND;
            IF bp.price < 3.00 THEN
                UPDATE Sells
                SET price = bp.price + 1.00
                WHERE bar = 'Joe''s Bar'
                    AND beer = bp.beer;
            END IF;
        END LOOP;
        CLOSE c;
    END;

    run
```