SQL3 Triggers

- Read in text.
- Some differences, including:
  1. Position of `FOR EACH ROW`.
  2. You can execute the action `INSTEAD OF` as well as before or after the event that triggered the action.
  3. The Oracle restriction about not modifying the relation of the trigger or other relations linked to it by constraints is not present in SQL3 (but Oracle is real; SQL3 is paper).
  4. The action in SQL3 is a list of SQL3 statements, not a PL/SQL statement.

SQL2 Assertions

- Database-schema constraint.
- Not present in Oracle 7.3.2.
- Checked whenever a mentioned relation changes.
- Syntax:

```
CREATE ASSERTION <name>
CHECK(<condition>);
```

Example

No bar may charge an average of more than \$5 for beer.

```
Sells(bar, beer, price)

CREATE ASSERTION NoRipoffBars
CHECK(NOT EXISTS(
          SELECT bar
          FROM Sells
          GROUP BY bar
          HAVING 5.0 < AVG(price)
     )
);
```

- Checked whenever `Sells` changes.

Example

There cannot be more bars than drinkers.

```
Bars(name, addr, license)
Drinkers(name, addr, phone)

CREATE ASSERTION FewBar
CHECK(
    (SELECT COUNT(*) FROM Bars) <=
    (SELECT COUNT(*) FROM Drinkers)
);
```

- Checked whenever `Bars` or `Drinkers` changes.

Challenge

Suppose we have our usual relations

```
Beers(name, manf)
Sells(bar, beer, price)
```

and we want to maintain the foreign-key constraint that if you sell a beer, its name must appear in `Beers`.

- Question: if we don't have foreign-key declarations available, how could we arrange for this constraint to be maintained:
  a) Using attribute-based constraints?
  b) Using SQL2 assertions?
  c) using Oracle triggers?
- Question: What if we *also* want to make sure that each beer mentioned in `Beers` is sold at at least one bar?

Another Constraints Problem

Recall our design of a DB for cities, counties and states, especially how we decided that we needed a relationship cities-states to support the "weakness" of cities. A DB schema might have:

```
States(name, pop, capital)
Cities(name, pop, state)
Counties(name, pop, state)
CinC(cityName, cityState,
    countyName, countyState)
```

- There is a funny form of redundancy, because we expect to be able to find the state of a city by going to *any* county for that city and finding the state of that county.
- Question: as written, is it possible for a city to be in counties that are in two different states? How would you enforce the requirement that that not happen?
- Question: if we are willing to enforce the constraint, can you simplify the database schema? Should you?

PL/SQL

- Found only in the Oracle SQL processor (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` 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.

- Form of declaration:

```
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 INOUT, as in ODL.
  - 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.

```
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:

```
BEGIN
    joeMenu('Bud', 2.50);
    joeMenu('MooseDrool', 5.00);
END;
.
run;
```

Assignment

Assign expressions to declared variables with `:=`.

Branches

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

- But in nests, use `ELSIF` in place of `ELSE IF`.

Loops

```
LOOP
        . . .
    EXIT WHEN <condition>
        . . .
END LOOP;
```

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.

```
Sells(bar, beer, price)
```

- This simple price-change algorithm can be implemented by a single `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
                UDPATE Sells
                SET price = thePrice + 1.00
                WHERE bar = 'Joe''s Bar'
                    AND beer = theBeer;
            END IF;
        END LOOP;
        CLOSE c;
    END;
.
run
```

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).

```
Sells(bar, beer, price)

DECLARE
    p Sells.price%TYPE;
BEGIN
    SELECT price
    INTO p
    FROM Sells
    WHERE bar = 'Joe''s Bar' AND
        beer = 'Bud';
END;
.
run
```