

Schedule

- Today: Feb. 5 (T)
 - ◆ Triggers, PL/SQL.
 - ◆ Read Sections 7.4, 8.2. Assignment 4 due.
- Feb. 7 (TH)
 - ◆ PL/SQL, Embedded SQL, CLI, JDBC.
 - ◆ Read Sections 8.1, 8.3-8.5.
- Feb. 12 (T) Advising Day. No class.
- Reminder: Midterm is Feb. 14 (TH)
 - ◆ Covers material through Feb. 7 (TH) lecture and readings (Chapters 1-3, 5-7, 8.1-8.5).
- Feb. 19 (T)
 - ◆ Object-Relational Systems.
 - ◆ Read Sections 4.5, 9.4-9.5. Assignment 5 due.

Modification to Views Via Triggers

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

Example

```
Likes(drinker, beer)  
Sells(bar, beer, price)  
Frequents(drinker, bar)
```

```
CREATE VIEW Synergy AS  
  SELECT Likes.drinker, Likes.beer,  
         Sells.bar  
  FROM Likes, Sells, Frequents  
 WHERE Likes.drinker = Frequents.drinker AND  
        Likes.beer = Sells.beer AND  
        Sells.bar = Frequents.bar;
```

```
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 Frequenters 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.

```
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;
```

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).

```
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

Functions (PostgreSQL Version)

Server-side functions can be written in several languages:

- SQL
- PL/PGSQL
- PL/TCL
- PL/Perl
- C

SQL Functions (PostgreSQL Version)

Like Oracle stored procedures.

`CREATE FUNCTION` requires the following information:

- Function name
- Number of function arguments
- Data type of each argument
- Function return type
- Function action
- Language used by the function action

Example

- A simple SQL function to convert a temperature from Fahrenheit to centigrade degrees.

```
CREATE FUNCTION ftoc(float)
RETURNS float
AS 'SELECT ($1 - 32.0) * 5.0 / 9.0;'
LANGUAGE 'sql';

SELECT ftoc(68);

      ftoc
-----
        20
(1 row)
```


Functions (Continued)

- SQL functions can return multiple values using SETOF.
- Function actions can also contain INSERTs, UPDATES, and DELETEs as well as multiple queries separated by semicolons.
- Arguments: \$1 is automatically replaced by the first argument of the function call. \$2 is the second argument, etc.

Example

SQL server-side function to compute a sales tax.

```
CREATE FUNCTION tax(numeric)
```

```
RETURNS numeric
```

```
AS 'SELECT ($1 *  
      0.06::numeric(8,2))::numeric(8,2);'  
LANGUAGE 'sql';
```

```
SELECT tax(100);
```

```
tax
```

```
-----
```

```
6.00
```

```
(1 row)
```

Notice the casts to NUMERIC(8,2) using the double-colon form of type casting, rather than CAST.

Server Side Functions in SQL Queries

```
CREATE TABLE part (
    part_id    INTEGER,
    name       CHAR(10),
    cost       NUMERIC(8,2),
    weight     FLOAT
);

INSERT INTO part VALUES (637, 'cable', 14.29, 5);
INSERT INTO part VALUES (638, 'sticker', 0.84, 1);
INSERT INTO part VALUES (639, 'bulb', 3.68, 3);

SELECT part_id, name, cost, tax(cost), cost+tax(cost) AS total
FROM part
ORDER BY part_id;
```

part_id	name	cost	tax	total
637	cable	14.29	0.86	15.15
638	sticker	0.84	0.05	0.89
639	bulb	3.68	0.22	3.90

(3 rows)

Example: Shipping

```
CREATE FUNCTION shipping(numeric)
RETURNS numeric
AS 'SELECT CASE
    WHEN $1 < 2          THEN CAST(3.00 AS numeric(8,2))
    WHEN $1 >= 2 AND $1 < 4 THEN CAST(5.00 AS numeric(8,2))
    WHEN $1 >= 4          THEN CAST(6.00 AS numeric(8,2))
END;'
LANGUAGE 'sql';
SELECT part_id, trim(name) AS name, cost, tax(cost),
       cost+tax(cost) AS subtotal, shipping(weight),
       cost+tax(cost)+shipping(weight) AS total
```

FROM part

ORDER BY part_id;

part_id	name	cost	tax	subtotal	shipping	total
637	cable	14.29	0.86	15.15	6.00	21.15
638	sticker	0.84	0.05	0.89	3.00	3.89
639	bulb	3.68	0.22	3.90	5.00	8.90

(3 rows)

Triggers (PostgreSQL Version)

Create a function for states that uses the new RECORD variable to perform the following actions:

- Reject a state code that is not exactly two alphabetic characters
- Reject a state name that contains nonalphabetic characters
- Reject a state name less than three characters in length
- Uppercase the state code
- Capitalize the state name

Example Function

```
CREATE FUNCTION trigger_insert_update_statename()  
RETURNS opaque  
AS 'BEGIN  
    IF new.code ! ''[A-Za-z][A-Za-z]${}''  
    THEN RAISE EXCEPTION ''State code must be two alphabetic  
        characters.'';  
    END IF;  
    IF new.name ! ''[A-Za-z ]*${}''  
    THEN RAISE EXCEPTION ''State name must be only alphabetic  
        characters.'';  
    END IF;  
    IF length(trim(new.name)) < 3  
    THEN RAISE EXCEPTION ''State name must longer than two  
        characters.'';  
    END IF;  
    new.code = upper(new.code); -- uppercase statename.code  
    new.name = initcap(new.name); -- capitalize statename.name  
    RETURN new;  
END;  
LANGUAGE 'plpgsql';
```

Trigger (PostgreSQL Version)

```
CREATE TRIGGER trigger_statename  
BEFORE INSERT OR UPDATE  
ON statename  
FOR EACH ROW  
EXECUTE PROCEDURE  
    trigger_insert_update_statename()  
;
```

Example Execution

```
INSERT INTO statename VALUES ('a', 'alabama');
ERROR: State code must be two alphabetic characters.
INSERT INTO statename VALUES ('al', 'alabama2');
ERROR: State name must be only alphabetic characters.
INSERT INTO statename VALUES ('al', 'al');
ERROR: State name must longer than two characters.
INSERT INTO statename VALUES ('al', 'alabama');
INSERT 292898 1
SELECT * FROM statename;
```

code	name
AL	Alabama

(1 row)

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
            UPDATE Sells
                SET price = thePrice + 1.00
                WHERE bar = 'Joe''s Bar'
                AND beer = theBeer;
        END IF;
    END LOOP;
    CLOSE c;
END;

```

•

run

Winter 2002

Arthur Keller – CS 180

10-27

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