Schedule

• Oct. 25 (TH) More Constraints, Triggers.
  ◆ Read Sections 7.3-7.4.

• Oct. 30 (T) Embedded SQL.
  ◆ Read Section 8.1.
  ◆ Assignment 5 due.
  ◆ Project Part 4 due.

• Nov. 1 (TH) Midterm
  ◆ Covers material through Oct. 25 (TH) lecture and readings (Chapters 1-3, 5-7).
Triggers (Oracle Version)

Often called event-condition-action rules.

- **Event** = a class of changes in the DB, *e.g.*, “insertions into Beers.”
- **Condition** = a test as in a where-clause for whether or not the trigger applies.
- **Action** = one or more SQL statements.
- Differ from checks or SQL assertions in that:
  1. Triggers invoked by the event; the system doesn’t have to figure out when a trigger could be violated.
  2. Condition not available in checks.
Example
Whenever we insert a new tuple into Sells, make sure the beer mentioned is also mentioned in Beers, and insert it (with a null manufacturer) if not.

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

CREATE OR REPLACE TRIGGER BeerTrig
AFTER INSERT ON Sells
FOR EACH ROW
WHEN(new.beer NOT IN
    (SELECT name FROM Beers))
BEGIN
    INSERT INTO Beers(name)
    VALUES(:new.beer);
END;

run
Options

1. Can omit OR REPLACE. But if you do, it is an error if a trigger of this name exists.

2. AFTER can be BEFORE.

3. If the relation is a view, AFTER can be INSTEAD OF.
   - Useful for allowing “modifications” to a view; you modify the underlying relations instead.

4. INSERT can be DELETE or UPDATE OF <attribute>.
   - Also, several conditions like INSERT ON Sells can be connected by OR.

5. FOR EACH ROW can be omitted, with an important effect: the action is done once for the relation(s) consisting of all changes.
Notes

• There are two special variables new and old, representing the new and old tuple in the change.
  ◆ old makes no sense in an insert, and new makes no sense in a delete.

• Notice: in WHEN we use new and old without a colon, but in actions, a preceding colon is needed.

• The action is a PL/SQL statement.
  ◆ Simplest form: surround one or more SQL statements with BEGIN and END.
  ◆ However, select-from-where has a limited form.
• Dot and \texttt{run} cause the definition of the trigger to be stored in the database.
  
  ◆ Oracle triggers are part of the database schema, like tables or views.

• Important Oracle constraint: the action cannot change the relation that triggers the action.
  
  ◆ Worse, the action cannot even change a relation connected to the triggering relation by a constraint, \textit{e.g.}, a foreign-key constraint.
Example

Maintain a list of all the bars that raise their price for some beer by more than $1.

Sells(bar, beer, price)
RipoffBars(bar)

CREATE TRIGGER PriceTrig
AFTER UPDATE OF price ON Sells
FOR EACH ROW
WHEN(new.price > old.price + 1.00)
BEGIN
    INSERT INTO RipoffBars
    VALUES(:new.bar);
END;
.
run
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 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.

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

\[
\text{CREATE PROCEDURE joeMenu(}
    \text{b IN Sells.beer \%TYPE,}
    \text{p IN Sells.price \%TYPE}
) \text{ AS}
\]

\[
\text{BEGIN}
    \text{INSERT INTO Sells}
    \text{VALUES('Joe''s Bar', b, p);}
\]

\[
\text{END;}
\]

\[
\text{run;}
\]

- 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.

```sql
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;
```

<table>
<thead>
<tr>
<th>part_id</th>
<th>name</th>
<th>cost</th>
<th>tax</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>637</td>
<td>cable</td>
<td>14.29</td>
<td>0.86</td>
<td>15.15</td>
</tr>
<tr>
<td>638</td>
<td>sticker</td>
<td>0.84</td>
<td>0.05</td>
<td>0.89</td>
</tr>
<tr>
<td>639</td>
<td>bulb</td>
<td>3.68</td>
<td>0.22</td>
<td>3.90</td>
</tr>
</tbody>
</table>

(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;

<table>
<thead>
<tr>
<th>part_id</th>
<th>name</th>
<th>cost</th>
<th>tax</th>
<th>subtotal</th>
<th>shipping</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>637</td>
<td>cable</td>
<td>14.29</td>
<td>0.86</td>
<td>15.15</td>
<td>6.00</td>
<td>21.15</td>
</tr>
<tr>
<td>638</td>
<td>sticker</td>
<td>0.84</td>
<td>0.05</td>
<td>0.89</td>
<td>3.00</td>
<td>3.89</td>
</tr>
<tr>
<td>639</td>
<td>bulb</td>
<td>3.68</td>
<td>0.22</td>
<td>3.90</td>
<td>5.00</td>
<td>8.90</td>
</tr>
</tbody>
</table>

(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

```sql
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;
<table>
<thead>
<tr>
<th>code</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>Alabama</td>
</tr>
</tbody>
</table>
(1 row)