Admin matters

- updated study guide
- essay presentations: 18 August (Topics 1, 2), 19 August (Topics 3, 4)

<table>
<thead>
<tr>
<th>Week</th>
<th>Date</th>
<th>Day</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>11 Aug</td>
<td>Tues</td>
<td>L7: Object-Relational DBMS</td>
</tr>
<tr>
<td></td>
<td>12 Aug</td>
<td>Wed</td>
<td>L8: Object-Relational DBMS</td>
</tr>
<tr>
<td></td>
<td>14 Aug</td>
<td>Fri</td>
<td><strong>Practical 3:</strong> Object-Relational DBMS (PostgreSQL)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Practical 4:</strong> XML DBs (BaseX)</td>
</tr>
<tr>
<td>5</td>
<td>18 Aug</td>
<td>Tues</td>
<td>L9: Semi-structured databases</td>
</tr>
<tr>
<td></td>
<td>19 Aug</td>
<td>Wed</td>
<td>L10: Semi-structured databases</td>
</tr>
<tr>
<td></td>
<td>21 Aug</td>
<td>Fri</td>
<td><strong>Practical 4:</strong> XML DBs (BaseX)</td>
</tr>
</tbody>
</table>
In this lecture

PostgreSQL advanced features

1. RECAP: functions
2. Triggers
**Summary of SQL:2008 continued**

<table>
<thead>
<tr>
<th>Activity</th>
<th>Statement example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special operations for changes to database data:</td>
<td>e.g.</td>
</tr>
<tr>
<td><strong>Trigger:</strong></td>
<td><strong>CREATE TRIGGER</strong> InsertPropTable AFTER INSERT ON PropertyForRent .....</td>
</tr>
<tr>
<td>SQL (compound) statement executed <em>automatically</em> by DBMS as <em>side effect</em> of a modification to named table</td>
<td></td>
</tr>
<tr>
<td><strong>Stored procedures</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Special data types</strong></td>
<td><strong>Character Large Object (CLOB)</strong> Binary Large Object (BLOB)**</td>
</tr>
</tbody>
</table>
• **SQL/PSM** *(ch. 8)*
  • SQL Persistent Stored Modules
  • Extensions to SQL that provide a (server-side) procedural language
    • used to write functions, stored procedures, Triggers that are stored on the DB server.

• **Databases** *(Relational / Object-Relational)* from different suppliers provide their own version of SQL/PSM e.g.
  • **ORACLE:** PL/SQL language *(ch. 8)*
  • **PostgreSQL:** PL/pgSQL
Four categories user defined functions:

1. query language functions (SQL functions: written in SQL)
2. procedural language functions (functions written in, e.g. PL/pgSQL or PL/Tcl)
3. internal functions
4. C-language functions

Two categories of SQL functions:

1. Row functions: operate on values of a table row
2. Table functions: operate on whole table
   - are functions that produce a set of rows, made up of:
     - either base data types (scalar types) or
     - composite data types (table rows).
CREATE FUNCTION functionName(argtype1, argtype2, .. ) RETURNS returntype AS
$$
SQL statement
$$ LANGUAGE SQL;

CREATE FUNCTION functionname(argtype1, argtype2, .. )
RETURNS returntype AS
$$
DECLARE --comment 1: declare vars if necessary
    var1 vartype1, var2 vartype2, .... ;
BEGIN -- comment 2: code to compute return value
RETURN returnvalue ; -- must be of returntype
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION functionname(...) RETURNS returntype AS $$

DECLARE var1 vartype1, var2 vartype2, .... ;
BEGIN

-- assignment statement
---1. e.g. var1 = 10; or var1 := 10;

-- programming constructs for conditional execution
--2. IF THEN END IF
--  IF THEN ELSE END IF
--  IF THEN ELSIF THEN END IF

--3. CASE varx WHEN END CASE
RETURN returnvalue ;
END;
$$ LANGUAGE plpgsql;
EXAMPLE: Given the table created as:

```sql
CREATE TABLE cities ( name text, population real, area int );
```

and function:

```sql
CREATE FUNCTION is_big_city(integer) RETURNS text AS $$
DECLARE   big text;
BEGIN
    IF $1 >= 1000 THEN
        big := 'YES';
    ELSE
        big := 'NO';
    END IF;
    RETURN big;
END;
$$ LANGUAGE plpgsql
```

and query:

```sql
SELECT name, area, is_big_city(area) FROM CITIES;
```
CREATE FUNCTION functionname(... ) RETURNS returntype AS $$
DECLARE var1 vartype1, var2 vartype2,.... ;
BEGIN

--programming constructs for looping

--4. LOOP EXIT WHEN END LOOP
--  LOOP EXIT WHEN CONTINUE WHEN END LOOP

--5. WHILE expr LOOP END LOOP

RETURN returnvalue ;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION functionname(... ) RETURNS returntype AS $$

DECLARE var1 vartype1, var2 vartype2, .... ;

BEGIN

--programming constructs for looping contnd

--6. FOR var IN listexpr LOOP END LOOP
-- FOR var IN REVERSE listexpr LOOP END LOOP

--7. FOREACH target IN ARRAY arrayexpr LOOP END LOOP
-- FOREACH target SLICE num IN ARRAY expression LOOP END LOOP

RETURN returnvalue ;

END;

$$ LANGUAGE plpgsql;
Given:
CREATE TABLE salEmp ( name  text,  payByQuarter int[ ], ..);

and function:
CREATE FUNCTION total_salary( int[ ] ) RETURNS int AS $$
DECLARE val int; total int;
BEGIN
    total = 0;
    FOREACH val IN ARRAY $1 LOOP
        total = total + val;
    END LOOP;
    RETURN total;
END;
$$ LANGUAGE plpgsql

and query:
SELECT name, paybyquarter, total_salary(paybyquarter)FROM SALEMP;
CREATE FUNCTION functionname( ... ) RETURNS returntype AS $$
DECLARE  var1 vartype1, var2 vartype2,.... ;
BEGIN
   --other constructs
   --8. SQL statement (YES!!!)
   RETURN returnvalue ;
END;
$$ LANGUAGE plpgsql;
A trigger defines an action that the DBMS should take when some event occurs.

Most commonly the event arises due to an operation (INSERT, UPDATE, or DELETE) on a database table.

A trigger may be defined as a BEFORE trigger or an AFTER trigger.

A BEFORE trigger executes before the specified operation while an AFTER trigger executes after the specified operation.

A trigger may be used to:
- enforce some referential constraints
- enforce complex constraints (business rules)
- audit changes to data
Triggers in PL/SQL (3b)

- (refer to chapter 8, section 8.3 of the textbook).
- Syntax of PL/SQL create trigger statement (similar to SQL/PSM standard):

```
CREATE TRIGGER  TriggerName
    BEFORE | AFTER | INSTEAD OF INSERT | DELETE | UPDATE
    [OF optional TriggerColumnList] ON TableName
    [REFERENCING {OLD | NEW}
        AS { OldName | newName } ]
    [FOR EACH { ROW | STATEMENT } ]
    [WHEN condition]
< trigger action>
```
Triggers in PL/SQL (3c)

- **Example** of PL/SQL create trigger statement:

```sql
CREATE TRIGGER StaffAfterInsert
    AFTER INSERT ON Staff
    REFERENCING NEW AS new
    FOR EACH ROW
    BEGIN
        INSERT INTO StaffAudit
        VALUES ( :new.staffNo, :new.fName, :new.lName);
    END;
```
CREATE FUNCTION triggerfunction() RETURNS TRIGGER AS $$
DECLARE -- declare vars if necessary
BEGIN
    -- code to compute any required values goes here.
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER triggername  -- e.g. of a before trigger
    BEFORE tableoperation ON tablename
    --- table operation is INSERT or DELETE or UPDATE
    FOR EACH ROW
    -- enables access to the NEW and OLD table rows
    EXECUTE PROCEDURE triggerfunction( );
Given the table created as:

```sql
CREATE TABLE cities ( name text, population real, area int );
```

```sql
CREATE FUNCTION check_valid_area() RETURNS TRIGGER AS $$
BEGIN
    IF (NEW.area <= 0) THEN
        RAISE EXCEPTION 'Invalid area value!';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```sql
CREATE TRIGGER valid_area
BEFORE INSERT OR UPDATE ON Cities
FOR EACH ROW
EXECUTE PROCEDURE check_valid_area();
```
Given the table created as:

```
CREATE TABLE cities ( name text, population real, area int );
```

```
SELECT * FROM Cities;
```

<table>
<thead>
<tr>
<th>name text</th>
<th>population real</th>
<th>area integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Johannesburg</td>
<td>2.5</td>
</tr>
<tr>
<td>2</td>
<td>Pretoria</td>
<td>1.5</td>
</tr>
</tbody>
</table>

```
INSERT INTO Cities VALUES ('Durban', 4.7, -3000);
```

```
ERROR:  Invalid area value!
```

```
INSERT INTO Cities VALUES ('Durban', 4.7, 3000);  
SELECT * FROM Cities;
```

<table>
<thead>
<tr>
<th>name text</th>
<th>population real</th>
<th>area integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Johannesburg</td>
<td>2.5</td>
</tr>
<tr>
<td>2</td>
<td>Durban</td>
<td>4.7</td>
</tr>
<tr>
<td>3</td>
<td>Pretoria</td>
<td>1.5</td>
</tr>
</tbody>
</table>
CREATE TABLE cities ( name text, population real, area int );

RECALL:
CREATE TABLE Capitals( state character(2)) INHERITS (cities);

SELECT * FROM Cities;

SELECT * FROM Capitals;

INSERT INTO Capitals VALUES ('Upington', 0.5, -1000, 'NC');
SELECT * FROM Cities;

***NOT GOOD***

***SOLUTION??***

-- PostgreSQL – Trigger example cntnd(4d)
CREATE TABLE cities ( name text, population real, area int );
CREATE TABLE Capitals( state character(2)) INHERITS (cities);

So,
CREATE TRIGGER valid_area
    BEFORE INSERT OR UPDATE ON Capitals
    FOR EACH ROW
    EXECUTE PROCEDURE check_valid_area();

DELETE FROM CAPITALS WHERE name = 'Upington';
INSERT INTO Capitals VALUES ('Upington', 0.5, -1000, 'NC');

SELECT * FROM Cities;
SELECT * FROM Capitals;

ERROR:  Invalid area value!
Alternatively the CREATE TRIGGER statement may specify the condition under which the trigger function (procedure) should be executed as follows:

```
CREATE TRIGGER triggername
    BEFORE tableoperation ON tablename
       --- table operation is INSERT or DELETE or UPDATE
    FOR EACH ROW
    WHEN (condition for function execution)
    EXECUTE PROCEDURE triggerfunction( );
```
A more detailed syntax of a trigger function (procedure) is as follows:

CREATE FUNCTION `triggerfunction()` RETURNS TRIGGER AS $$

DECLARE
    -- comment 1: declare vars if necessary var1 vartype1, var2 vartype2,... ;
BEGIN
    -- comment 2: code to compute any required values goes here. In the code,
    -- use NEW.attribute1,... to access values of new input row
    -- e.g for INSERT use OLD.attribute1,...to access values of existing row in table
    -- e.g for UPDATE, DELETE
    IF (condition to test for unwanted situation) THEN
        RAISE EXCEPTION 'error message for exception';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;