Lecture 10
Real SQL Application Programming (Part 1)

Instructor: Shel Finkelstein

Reference:
A First Course in Database Systems,
3rd edition, Chapter 9

Important Notices

- Lecture slides and other class information will be posted on Piazza (not Canvas).
  - Slides are posted under Resources→Lectures
  - Lecture Capture recordings are available to all students under Yuja.
    - That includes classes given over Zoom.
  - There's no Lecture Capture for Lab Sections.
- All Lab Sections (and Office Hours) normally meet In-Person, with rare exceptions announced on Piazza.
  - Some slides for Lab Sections have been posted on Piazza under Resources→Lab Section Notes
- Office Hours for TAs and me are posted in Syllabus and Lecture1, as well as in Piazza notice @7; that post also now includes hours for Group Tutors.
- Some suggestions about use (and non-use) of Generative AI systems such as ChatGPT were posted on Piazza in @41, with specific discussion about the Movies tables and the four Avatar queries in Lecture 4.

Important Notices

- CSE 180 Midterm was **on Wednesday, February 14.**
  - Midterm and Midterm answers were posted on Piazza under Resources→Exams on Wednesday, February 14.
  - Answers were discussed at the beginning of class on Friday, February 16.
- Midterm grades were unmuted on Tuesday, February 20.
  - Students can see submissions, scores and comments by accessing assignments on Canvas, which will take them to GradeScope.
- Median score was 83.5. There is no curve for the Midterm.
  - Maximum possible score was 101, but grades are treated as if out of 100. For example, 83.5 is 83.5/100 (83.5%), not 83.5/101.
- Ask your TA about your Midterm grade if you have questions about scoring, and then contact me if you still have questions.
  - You cannot request regrades within GradeScope.
  - Deadline for asking questions about Midterm grade is Monday, February 26.

Important Notices

- Lab3 was posted on Piazza on Wednesday, February 14.
  - Lab3 is due on **Tuesday, February 27**.
    - Lab3 has <u>many</u> parts, and some parts of Lab3 are difficult.
    - You'll had enough information to do Lab3 after the class on on Friday, February 16.
      - You don’t need to use Triggers for Lab3.
  - You <u>must</u> use the Lab3 create file and load_lab3.sql file to do Lab3.
    - This <u>original</u> load data must be reloaded at <u>multiple</u> stages of Lab3.
    - The create_lab3.sql file was edited on Thursday, February 15 at about noon.
      - If you downloaded it before that, please download it again.
- To make Section 2.6.1 of Lab3 (createview.sql) simpler, I've modified the description of createview.sql on Tuesday, February 20 to include following line near the end:
  - For Lab3, you should assume that there's at least one role for every actor, so calculatedRowCount will be at least 1.

Important Notices

- The Fourth Gradiance Assignment, CSE 180 Winter 2024 #2, was assigned on Friday, February 23.
  - You should have enough information to complete this Gradiance Assignment after the lecture on Friday, February 23.
  - It is due on Saturday, March 2, by 11:59pm (yes, on Saturday, not Sunday).
  - Some of the 9 Gradiance questions are difficult.
- The subject of Lab4 is Real Application Programming, using libpq (next Lecture) and a Stored Function (this Lecture).
  - Lab4 will be posted on Piazza on Wednesday, February 28.
  - Lab4 is hard, and many students will need help completing it

Important Notices: Final

- CSE 180 Final is on **Wednesday, March 20, 8:00-11:00am, and it will be given in-person in our classroom, except for students who receive Remote Exam permission.**
  - **No** early/late Exams. **No** make-up Exams. **No** devices.
  - 3 hours, extended for DRC students, covering the entire quarter.
    - Final will be harder than the Midterms.
  - **During the Final, you may use one double-sided 8.5 x 11 sheet (with anything that you want written or printed on it).**
    - (Okay, you may bring two sheets with writing on only one side of each sheet.)
  - If you're ill, you'll need to request Remote Permission via email with Subject "Taking CSE 180 Final Remotely", sent before 6:00pm on the day before the Final.
    - Students who receive Remote Permission must be on Zoom during Exam.
    - Ask me again, if you haven't received a reply to your Remote Permission request!
  - Write your answers on the exam itself, readably, using **either ink or a #2 pencil**.
  - We will assign seats as you enter the room. You may not choose your own seat.
  - You must show your UCSC ID at the end of the exam.
  - Winter 2023 CSE 180 Final will be posted on Piazza by Monday, March 4.

A Word to the Unwise

- This is a tough class for some students since it involves a combination of theory and practice.
  - The second half of CSE 180 is much harder than the first half of the course.
- Students who regularly attend Lectures and Lab Sections (and Office Hours and Tutoring) often do well; students who don't regularly attend often do poorly.
  - We don't take attendance; you're responsible for your own choices.
- After the course ends, your course grade will be determined by your scores on Exams, Labs and Gradiance, as described on the "Course Evaluation" and "Grading" Slides in the Syllabus.
  - You won't be able to do any additional work to improve your grade.

SQL in Real Programs

- We have seen only how SQL is used at a generic query interface --- an environment where we sit at a terminal and ask queries of (or modify) a database.
- Reality is almost always <u>different</u>!
    - Conventional programs written in C or Java (or other languages) that interact with database using SQL.
    - Why?

Application Programming Approaches

1. Code in a specialized Stored Procedure language is stored in the database, and it is executed in the database when the Procedure is called.
   - **Stored Procedure** languages such as Persistent Stored Modules (PSM "standard"), PL/SQL (Oracle) and PL/pgSQL (PostgreSQL).
2. SQL statements are **embedded in a host language** (e.g., C).
   - Less important today than in early relational applications.
3. **Connection tools/libraries** are used to allow a conventional language to access a database.
   - CLI (Call-Level Interface) for C, which is based on ODBC (Open Database Connectivity)
     - CLI is SQL standard, but ODBC (Microsoft) is more commonly used.
   - libpq, which is a "native" PostgreSQL interface to PostgreSQL libraries.
   - JDBC (Java Database Connectivity) for Java, which has similarities to ODBC.
4. RESTful Web Services using HTTP (not in this Lecture or in our textbook).
   - See PostgreSQL RESTful API and HTTP API for PostgreSQL proposals.

Approach 1: Stored Procedures (and Functions)

- PSM, or "Persistent Stored Modules," allows us to store procedures as database schema elements.
  - Stored Procedures/Stored Functions are executed within the DB.
- PSM is a mixture of conventional programming language constructs (IF, WHILE, etc.) and SQL.
- PSM lets us do things we cannot do in SQL alone.
  - SQL is not a complete programming language.
  - But you still can't do "everything" in PSM, since PSM execution is within the DB.
    - Examples of what you can't do?

Alternatives to PSM

- Oracle uses PL/SQL, a variation of SQL/PSM that helped inspire PSM.
    - PL/SQL not only allows you to create and store procedures or functions, but it also can be run from Oracle's *generic query interface (SQL*Plus)*, just like any SQL statement.
- PostgreSQL: PL/pgSQL **(needed for Lab4)**
    - PostgreSQL now has Stored Procedures as well as Stored Functions.
        - For Lab4, you'll only need a Stored Function, not a Stored Procedure.
    - PL/pgSQL Stored Procedure and Function calls can be executed from psql
- IBM DB2: SQL PL
- MS SQL Server and Sybase: Transact-SQL (T-SQL)

Info on PL/pgSQL

Every database system (including PostgreSQL) has its own extensions and variations for SQL, and PostgreSQL has its own version of Stored Procedures Stored Functions, **PL/pgSQL**, which resembles (but is not the same as) Oracle's PL/SQL.

- The PostgreSQL manual has a Guide for defining stored procedures/functions.
  - The PostgreSQL manual page on PL/pgSQL syntax and structure also will help you a lot.
- This PostgreSQL Tutorial site has focused information about PL/pgSQL that's more readable than the PostgreSQL manual.

Basic PL/pgSQL Form

```
CREATE PROCEDURE <name> (<parameter list>)
    AS $$
     <optional_local_declarations>
     <body>
    $$ LANGUAGE plpgsql;


CREATE FUNCTION <name> (<parameter list>) RETURNS <type>
    AS $$
     <optional_local_declarations>
     <body>
    $$ LANGUAGE plpgsql;
```

Parameter Modes

- Unlike the usual name-type pairs in languages like C, parameters can be mode-name-type triples, where ***mode*** can be:
  - IN mode: Value can be used, and value can not be changed.
    - In PL/pgSQL, this is the default mode
  - OUT mode: Value can not be used, and value can be changed.
  - INOUT mode: Value can be used, and value can be changed
- Function parameters may have any of these three modes.
- Procedure parameters can have mode IN or mode INOUT; they can not have mode OUT.
- (Other Stored Procedure languages have different restrictions on mode.)

Example: Stored Procedure

- Let's write a procedure JoeMenu that takes two arguments *b* and *p*, and adds a tuple to Sells(bar, beer, price) that has:

  bar = 'Joe''s Bar',

  beer = *b*, and

  price = *p*.

- Used by Joe to add to his menu more easily.

The Procedure

```
CREATE PROCEDURE JoeMenu (
    IN    b    CHAR(20),
    IN    p    REAL
)
    AS $$

  INSERT INTO Sells
     VALUES('Joe''s Bar', b, p);

$$ LANGUAGE plpgsql;
```

Parameters are both read-only, not changed

The body --- a single insertion

Invoking Stored Procedures and Stored Functions

- Use the PL/pgSQL statement CALL, with the name of the desired Stored Procedure and the argument values for that procedure.

  Example:

  ```
  CALL JoeMenu('Moosedrool', 5.00);
  ```

- A Stored Function may be used in SQL expressions <u>wherever a value of its return type could appear</u>.
  - **Composition** yet again!

- PostgreSQL Stored Procedures can explicitly execute transactions, but Stored Functions can't.
  - But Stored Functions can be invoked from within transactions.

Kinds of PL/pgSQL statements (1)

- RETURN; returns from Stored Procedure.
- RETURN <expression>; returns from a Stored Function, returning the value of that expression to its invoker.

- DECLARE <name> <type> [:= <initial value>];
  is used to declare (and initialize) local variables.
  - A DECLARE may be followed by any number of
    <name> <type>; pairs.
  - If there's no initial value for a local variable, then that variable's initial value is NULL.
  - SQL types, such as INTEGER, DATE, NUMERIC and CHAR can be used in Stored Function declaration of parameters and local variables.

Kinds of PL/pgSQL Statements (2)

- BEGIN . . . END is used to group statements, such as the statements in a function or a procedure.
  - Separate those statements with semicolons.

- Assignment statements:
  <variable> := <expression>;
  - Example: b := 'Bud';

- Statement labels: You can give a statement (such as a LOOP statement) a label by prefixing a name and a colon.

IF Statements

- Simplest IF statement:

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

Can test for NULL in PL/pgSQL IF statements conditions using IS NULL and IS NOT NULL.

- Add ELSE <statement(s)> if desired:

```
        IF . . . THEN . . . ELSE . . . END IF;
```

- Add additional cases by ELSIF <statement(s)>:

```
IF … THEN … ELSIF … THEN … ELSIF …
        THEN … ELSE … END IF;
```

Example: IF

- Let's rate bars by how many customers they have, based on Frequents(drinker,bar).
  - < 100 customers: 'unpopular'.
  - 100-199 customers: 'average'.
  - >= 200 customers: 'popular'.
- Function Rate(b) rates bar b.

Example: IF (continued)

```
CREATE FUNCTION Rate (IN b CHAR(20) )
          RETURNS CHAR(10)
 AS $$
    DECLARE cust INTEGER;
    BEGIN
        cust := (SELECT COUNT(*) FROM Frequents
                        WHERE bar = b);
        IF cust < 100 THEN RETURN 'unpopular';
        ELSIF cust < 200 THEN RETURN 'average';
        ELSE RETURN 'popular';
        END IF;
    END;
$$ LANGUAGE plpgsql;
```

Local variable

Number of customers of bar b

Nested IF statement

CASE Statement with Expression

CASE statement can be based on value of an Expression, or based on a Condition.
Example with an **Expression**:

```
CASE <expression>
        WHEN <expression value> THEN
                <statements>
        [ WHEN <expression value> THEN
                <statements> ]
               [ … ]
        [ ELSE
                <statements> ]
END CASE;
```

Only the statements for first expression value matched are executed.

Multiple value expressions can appear for each WHEN.
Separate them with commas.

Good description, with examples, is here.
Example of CASE with Condition is on the next slide.

CASE Statement with Condition

```
CREATE FUNCTION Rate (IN b CHAR(20) )
            RETURNS CHAR(10)
 AS $$
    DECLARE
            cust INTEGER;
            answer CHAR(10);
    BEGIN
            cust := ( SELECT COUNT(*) FROM Frequents
                        WHERE bar = b );
            CASE
                WHEN cust < 100 THEN
                     answer := 'unpopular';
                WHEN cust < 200 THEN
                     answer := 'average';
                ELSE
                     answer :=  'popular';
            END CASE;
            RETURN answer;
    END;
$$ LANGUAGE plpgsql;
```

Only statements for first condition matched are executed.

Loops

- Basic form:

```
<loop name>: LOOP
        <statements>
        END LOOP;
```

- Exit from a loop by:

```
 EXIT [<loop name>] [WHEN condition];
```

   where condition is any boolean expression on variables and constants.

   - Loop exits if condition is TRUE.

Example: Exiting a Loop

```
loop1: LOOP
        . . .
        EXIT loop1 WHEN <condition>;
                                    ←——— If condition is TRUE . . .
        . . .
    END LOOP;
    ←——————————— Control winds up here
```

- Loops can be nested; EXIT <loop label> exits that loop.
  - EXIT without a <loop label> exits innermost loop.
- Can use CONTINUE instead of EXIT to continue execution of loop at its start, including with label and condition.

Other Loop Forms

```
WHILE <condition> DO
        <statements>
 END WHILE;

FOR <variable> IN <range> LOOP
        <statements>
 END LOOP;
```

- CONTINUE and EXIT can also be used with these loops.
- <range> could be something like 1..10
  …but we'll see more about FOR LOOP later.

Queries

- General SELECT-FROM-WHERE queries are *not* permitted in Stored Procedure Languages such a PL/pgSQL.
- There are three ways to get the results of a query:
    1. Queries which produce one scalar value can be the expression in an assignment.
    2. Single-row SELECT . . . INTO …
    3. Cursors

Example: Assignment/Query

- Using local variable *p* and Sells(bar, beer, price), we can get the price Joe charges for Bud by:

```
p := ( SELECT price FROM Sells
        WHERE bar = 'Joe''s Bar'
          AND beer = 'Bud' );
```

SELECT . . . INTO …

- You can get the value of a query that returns one tuple is by placing INTO <variable> after the SELECT clause.

- Example with single attribute in the SELECT clause:

```
SELECT price INTO p
FROM Sells
WHERE bar = 'Joe''s Bar'
    AND beer = 'Bud';
```

But what if we wanted to SELECT both price and quantity (assuming Sells had a quantity) attribute?

SELECT . . . INTO … (continued)

- You can get the value of a query that returns one tuple is by placing INTO <variable> after the SELECT clause.

- Example with multiple attributes in SELECT clause:

```
 SELECT price, quantity INTO p, q
 FROM Sells
 WHERE bar = 'Joe''s Bar'
     AND beer = 'Bud';
```

If we wanted to SELECT both price and quantity (assuming Sells had a quantity) attribute).

Cursors

- A *cursor* is essentially a tuple-variable that ranges over all tuples in the result of some query.

- Declare a cursor *c* by:
  DECLARE c CURSOR FOR <query>;

- To use cursor *c*, we must issue the command:
  OPEN c;
  - The query in the declaration of *c* is evaluated, and cursor *c* is set to point to the first tuple of the result.
  - When you DECLARE a cursor, the statement associated with the cursor can reference local variables. Those local variables are bound to their current values only when you OPEN the cursor.

- When finished with *c*, issue the command:
  CLOSE c;

Fetching Tuples From a Cursor

- To get the the tuple that cursor c currently points to, issue the command:

  FETCH FROM c INTO *x1, x2,…, xn* ;

- The *x*'s are a list of variables, one for each attribute in the SELECT clause of the query that's in the declaration of *c*.

- After FETCH is executed, cursor c is moved automatically to point to the <u>next tuple</u> in the result of that query.

Breaking Cursor Loops – (1)

- The usual way to use a cursor is to create a loop that has a FETCH statement in it, and that does something with each tuple that is fetched.

- A tricky question:

  How we get out of the loop when the cursor has no more tuples to deliver?

Breaking Cursor Loops – (2)

The PostgreSQL manual section on Cursors describes the use of cursors in PL/pgSQL.

- After a PL/pgSQL FETCH, you can use a special variable FOUND (which you don't declare) to test whether or not the FETCH returned a tuple.
- So the way to EXIT a loop after a FETCH, when no tuple is found, is by saying:

```
EXIT WHEN NOT FOUND;
```

The <Condition> for EXIT can be any boolean expression using variables, constants, and FOUND.

Breaking Cursor Loops – (3)

- The structure of a cursor loop could be:

```
cursorLoop:
   LOOP
          …
          FETCH c INTO x1, x2, …, xn;
          EXIT WHEN NOT FOUND;
          …
    END LOOP;
```

Breaking Cursor Loops – (4)

- The structure of a cursor loop could also be:

```
    cursorLoop:
        FOR x1, x2, …, xn IN c LOOP
                …
        END LOOP;
```

- EXIT and CONTINUE (and labels) can be used with FOR loops, just as with other loops.
    - You can also write an explicit query, instead of using cursor c.
- This FOR LOOP construct only works in PL/pgSQL when there is just a <u>single</u> variable x1 (corresponding to a <u>single</u> attribute) in cursor c, <u>not</u> when there is more than one variable returned by the cursor.
- To handle multiple variables, we could declare a PL/pgSQL variable of type <u>RECORD</u>, instead of using x1, x2, …, xn
    - We don't illustrate that in this Lecture.

Example: fireSomePlayersFunction

What does the example Stored Function *fireSomePlayersFunction* do?

Tables mentioned below (from a previous quarter's database schema):

- Persons(personID, name, address, salary, canBePlayer, canBeCoach)
- Teams(teamID, name, coachID, teamCity, teamColor, totalWins, totalLosses)
- Players(playerID, teamID, joinDate, rating, isStarter)
- Games(gameID, gameDate, homeTeam, visitorTeam, homePoints, visitorPoints)
- GamePlayers(gameID, playerID, minutesPlayed)

This schema is shown only to help you understand what the Stored Function **fireSomePlayersFunction** does. For Lab4, you won't have the tables and load data necessary to run this Stored Function.

What's an ”At Risk” Player?

The Stored Function *fireSomePlayersFunction* has one integer parameters, maxFired.

For each fired player, *fireSomePlayersFunction* will change the teamID for that player's tuple in the Players table to NULL. This Stored Function should count the number of the players who are fired, and it should return that fired count.

The fired count returned will never be more than maxFired. (It might equal maxFired, but it might also be less than maxFired.)

Here's how to determine which players should be fired. rating is an attribute in the Players table. minutesPlayed is an attribute in the GamePlayers table. We'll say that a player is “at risk” if the following three things <u>all hold for that player</u>:

- the player's team isn't NULL (so they're currently on a team), and
- the player's rating is 'L' (low player rating) and
- the player's total minutes played (adding up their minutesPlayed in GamePlayers) is more than 60.

Which “At Risk” Players are Fired?

Some of the “at risk” players will be fired … but some won't be fired. How do you decide which “at risk” players will be fired? salary is an attribute in the Persons table.

You'll look at the “at risk” players, and you'll fire the maxFired players who have the highest salaries.

Examples showing how *fireSomePlayersFunction* should work:

- If there are 10 “at risk” players and maxFired is 6, you'll fire the 6 “at risk” players who have the highest salary. If the players with the sixth and seventh highest salary have exactly the same salary, it's okay to fire either one (but only fire one). The fired count value returned will be 6.
- If there are 10 “at risk” players and maxFired is 10, you'll fire all 10 “at risk” players, and the fired count value returned will be 10.
- But if there are 10 “at risk” players and maxFired is 12, you'll fire all 10 “at risk” players, and the fired count value returned will be 10, not 12.

1-CREATE and Local Variables

```
CREATE OR REPLACE FUNCTION
 fireSomePlayersFunction(maxFired INTEGER)
      RETURNS INTEGER AS $$

   DECLARE
      numFired      INTEGER;
                          /* Number actually fired, the value returned */
      thePlayerID   INTEGER;  /* The player to be fired */
```

2-DECLARE CURSOR

```
DECLARE firingCursor CURSOR FOR
      SELECT p.personID
       FROM Persons p, Players play, GamePlayers gp
       WHERE p.personID = play.playerID
           AND play.playerID = gp.playerID
           AND play.rating = 'L'
           AND play.teamID IS NOT NULL
       GROUP BY p.personID
       HAVING SUM(gp.minutesPlayed) > 60
       ORDER BY p.salary DESC;
```

3-BEGIN and Validation

```
BEGIN

  -- Input Validation
  IF maxFired <= 0 THEN
    RETURN -1;              /* Illegal value of maxFired */
    END IF;

  numFired := 0;
            /* Could be initialized in declaration of numFired */

  OPEN firingCursor;
```

4-Firing LOOP

```
LOOP

    FETCH firingCursor INTO thePlayerID;

    -- Exit if there are no more records for firingCursor,
    -- or when we already have performed maxFired firings.
    EXIT WHEN NOT FOUND OR numFired >= maxFired;

    UPDATE Players
    SET teamID = NULL
    WHERE playerID = thePlayerID;

    numFired := numFired + 1;

END LOOP;
```

5-Finish and RETURN

```
CLOSE firingCursor;

        RETURN numFired;

    END

$$ LANGUAGE plpgsql;
```

The Needed Declarations

(in order to UPDATE/DELETE using CURRENT of Cursor, as on next slide)

```
CREATE PROCEDURE JoeGouge( )
    DECLARE theBeer CHAR(20);
    DECLARE thePrice REAL;
    DECLARE NotFound CONDITION FOR
     SQLSTATE '02000';
    DECLARE c CURSOR FOR
       SELECT beer, price FROM Sells
      WHERE bar = 'Joe''s Bar'
         FOR UPDATE;
```

The Procedure Body:
Using CURRENT OF Cursor

```
BEGIN
   OPEN c;
   menuLoop: LOOP
        FETCH c INTO theBeer, thePrice;
        IF NotFound THEN LEAVE menuLoop END IF;
        IF thePrice < 3.00 THEN
            UPDATE Sells SET price = thePrice + 1.00
            WHERE CURRENT OF c;
        END IF;
   END LOOP;
   CLOSE c;
END;
```

Check if the recent FETCH failed to get a tuple

If Joe charges less than $3 for the beer, raise its price at Joe's Bar by $1.

There are complex subtleties that affect use of CURRENT OF Cursor.
I suggest that you avoid using it until you learn those subtleties.

HOW MANY ROWS WERE CHANGED?

After you execute a UPDATE, DELETE or INSERT statement in PL/pgSQL, you can find out how many tuples were modified by saying:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

That will set the variable integer_var to be the number of rows that were affected by the previous SQL statement.

(There's nothing special about the name integer_var.)

For example, after you execute an UPDATE in a Stored Function, you can find out how many tuples were updated.

You can get more information about GET DIAGNOSTICS here.

Using a pgsql file

- Write the code for your Stored Function (or Stored Procedures) in a textfile such as myStoredFunction.pgsql

  After you've saved that file, you can create the Stored Function myStoredFunction that is created in that file by issuing the psql command:

  ```
  \i myStoredFunction.pgsql
  ```

- You can now invoke that function from your C program or from psql by executing:

  ```
  SELECT myStoredFunction(5);
  ```

  assuming that myStoredFunction has an integer argument.

- But you can also use myStoredFunction(5) in a SQL statement anywhere you can use the type that myStoredFunction returns!

Debugging Stored Functions (and Stored Procedures)

It would be very difficult to debug Stored Functions if you couldn't print out messages during testing. PL/pgSQL can print errors and messages using RAISE.

RAISE syntax somewhat resembles that for C input/output, except that there are no parentheses, and the optional parameters (corresponding to appearances of %) must be strings.

- Use CAST, as shown below, to cast a different type to string.

Examples:

```
RAISE NOTICE  'Calling CreateJob for %', job_id;

RAISE NOTICE  'myInteger parameter is %', CAST( myInteger AS CHAR(10) );

RAISE EXCEPTION  'Person with first name % and last name % not found', firstName, lastName;
```

RAISE NOTICE prints an information message and continues execution. RAISE EXCEPTION raises an error and does not continue execution.

In the Lab4 solution that you submit after you've debugged your Stored Function, all output should be done from your C program, not by using RAISE in the Stored Function, which you should only use for debugging.

More on PL/pgSQL (for Lab4)

- Lab4 will be posted on Piazza under Resources→Lab4 on Wednesday, February 28, and it will be due on Tuesday, March12 by 11:59pm.
  - Resources→Lab4 will have the same PL/pgSQL Stored Function fireSomePlayersFunction example that we just discussed, as well the same file explaining what fireSomePlayersFunction does.
  - Lab4 is the one of the hardest Lab Assignments, so please start it <u>early</u>!
- PL/pgSQL has a bunch of differences from the PSM standard.
  - Having a SQL standard is wonderful in many ways, but:
    - The SQL standard doesn't specify everything.
    - Every system has extensions to the SQL standard.
    - Many systems deviate in some ways from the SQL standard.
- Some of the <u>many</u> significant ways that PL/pgSQL differs from PSM:
  1) Different syntax for creating Stored Procedures and Functions
  2) Different way of handling “Not Found” for loops in PL/pgSQL
- You should know what PSM is, but you <u>don't</u> have to know how to write PSM, since it hasn't been taught in Lecture.

Triggers and Stored Procedures

Trigger

- *Event* : typically a type of database modification, e.g., "insert on Sells"
- *Condition* : Any SQL boolean-valued expression
- *Action* : Any SQL statements

- Triggers may invoke Stored Procedures.
- A typical trigger body (actions) may itself be thought of as an unnamed Stored Procedure.
  - In PostgreSQL, a Trigger's action <u>must</u> be a Stored Procedure.
    - EXECUTE PROCEDURE function_name (arguments)
  - In some systems, the Trigger body may include many of the kinds of statements that can be in a Stored Procedure.

Application Programming Approaches

1. Code in a specialized Stored Procedure language is stored in the database, and it is executed in the database when the Procedure is called.
   - **Stored Procedure** languages such as Persistent Stored Modules (PSM "standard"), PL/SQL (Oracle) and PL/pgSQL (PostgreSQL).
2. SQL statements are **embedded in a host language** (e.g., C).
   - Less important today than in early relational applications.
3. **Connection tools/libraries** are used to allow a conventional language to access a database.
   - CLI (Call-Level Interface) for C, which is based on ODBC (Open Database Connectivity)
     - CLI is SQL standard, but ODBC (Microsoft) is more commonly used.
   - libpq, which is a "native" PostgreSQL interface to PostgreSQL libraries.
   - JDBC (Java Database Connectivity) for Java, which has similarities to ODBC.
4. RESTful Web Services using HTTP (not in this Lecture or in our textbook).
   - See PostgreSQL RESTful API and HTTP API for PostgreSQL proposals.

Application Programming Approaches

When and Why do you use each?

1. **Stored Procedure** languages, such as PSM, PL/SQL and PL/pgSQL.
   - What are the advantages of using Stored Procedures and Functions?
2. SQL statements **embedded in a host language** (e.g., C).
3. **Connection tools/libraries** such as CLI, libpq and JDBC.
4. Restful Web Services using HTTP.