Lecture 11
Real SQL Application Programming (Part 2)

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

- Lab3 scores were unmuted on Monday, March 4.
  - Deadline for asking questions about your score is Friday, March 8.
- The Fifth Gradiance Assignment, CSE 180 Winter 2024 #5, will be assigned after you've learned about Design Theory (Lectures 12 and 13).
  - The Fourth Gradiance Assignment, CSE 180 Winter 2024 #4 was due on Saturday, March 2, by 11:59pm.
- Explanation of two difficult Gradiance questions on transactions was posted on Piazza on Sunday, March 2 in @122.
- Answers for two of the "Practice" Relational Algebra queries (in magenta) were posted on Piazza on Sunday, March 2 in @126.
- Winter 2024 Student Experience of Teaching Surveys - SETs open on Monday, March 4.
  - SETs close on Sunday March 17 at 11:59pm.
  - Instructors are not able to identify individual responses.
  - Constructive responses help improve future courses.

Important Notices

- The subject of Lab4 is Real Application Programming, using libpq (this Lecture) and a Stored Function (previous Lecture).
  - Lab4 was posted on Piazza on Wednesday, February 28.
  - Lab4 is hard, and many students will need help completing it.
    - Read the Lab4 announcement on Piazza as soon as possible!
    - We provided information files and examples, posted under Resources→Lab4, and described in the Lab4 pdf and announcement.
    - We have also provided load data that you can use to test your Lab4 solution.
      - But testing can prove that program is incorrect, not that it's correct.
  - Lab4 is due on **Tuesday, March 12 by 11:59pm**.
    - unix.ucsc.edu gets busy near the end of the quarter.
    - Please avoid Infinite Loops in your Stored Function.
  - All material needed for Lab4 was covered in Lecture by Friday, March 1.
  - However, there were some important clarifications and corrections which were made to Lab4.
    - These corrections, which are described in @121 on Piazza, "Important updates of multiple Lab4 files", are summarized on the next slide.

Important Updates of Multiple Lab4 Files Slide #1 (see @121)

- The Lab4 pdf is now correctly called CSE180_W24_Lab4.pdf
- A new version of the load file, load_lab4.sql, has ben posted.
- Several changes have been made in the runShakespeareApplication.c skeleton file.
  - The signature for the C function IncreaseSomeCastMemberSalaries in the runShakespeareApplication.c skeleton has been changed to:

    ```
    float increaseSomeCastMemberSalaries(PGconn *conn, char *thePlayTitle,
                int theProductionNum, float maxDailyCost)
    ```

    - That is, the maxDailyCost parameter is float, and the function returns a float value.
  - Minor: The name of the function begins with a lowercase "i", just as all the other C functions begin with lowercase letters.
    - Same is true for the Stored Function which is invokes, which is called increaseSomeCastMemberSalariesFunction (lowercase i, not uppercase).
  - Minor: Parameter which had been called theProdNum is now called theProductionNum, aligning with productionNum attribute name in tables.

Important Updates of Multiple Lab4 Files Slide #2 (see @121)

- The first lines of the Stored Function increaseSomeCastMemberSalariesFunction (which we did not give you in the Lab4 pdf) should be:

```
CREATE OR REPLACE FUNCTION
    increaseSomeCastMemberSalariesFunction(thePlayTitle VARCHAR(40),
            theProductionNum INTEGER, maxDailyCost NUMERIC(7,2))
 RETURNS NUMERIC(7,2) AS $$
```

  - Note that C does not have a type corresponding to NUMERIC(7,2).

- The value returned by increaseSomeCastMemberSalariesFunction when you execute that Stored Function in your C program (and then use PQgetvalue(res,0,0) to get get the result of the Stored Function) is a string.
  - Just as you can convert an appropriate character string to an integer using C's atoi function, you can convert a character string to a float using C's atof function, and increaseSomeCastMemberSalaries needs to return a float.

- The value that you print out in your tests of increaseSomeCastMemberSalaries should be a number with 2 decimal places. You can print out a floating point value in that format using the %7.2f format string (instead of using %f or %d).

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 <u>much harder</u> 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.

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 2: Embedded SQL

- Key idea: A pre-processor turns SQL statements into procedure calls that fit with the surrounding host-language code.
- All embedded SQL statements begin with EXEC SQL, so the pre-processor can find them easily.
- Approach 2 isn't frequently used any more.
  - The pre-processor step in Approach 2 makes compilation and debugging complex.
  - There are good, easy-to-use and easy-to-understand libraries for database access (Approach 3).
  - The pre-processor maps an Approach 2 program into an Approach 3 program, so why not go directly to Approach 3?

Approach 3: Host Language/SQL Interfaces via Libraries

- The third approach to connecting databases to conventional languages is to use library calls.

1. CLI is SQL standard, but ODBC (Microsoft) is more commonly used.
2. libpq, which is a “native" PostgreSQL interface to PostgreSQL libraries.
3. JDBC (Java Database Connectivity) for Java, which has similarities to ODBC.

Three-Tier Architecture

- A common environment for using a database had three tiers of processors:

1. *Web Servers* --- Interact with user, other Web Servers and Database Servers.
2. Application Servers --- Execute Business Logic.
   - Not separate servers any more.
   - Instead, Business Logic is usually in Web Servers, frequently accessing a collection of “Microservices".
3. *Database Servers* --- Get what other Servers need from the database.

Example: Product Purchase

- Database holds the information about products, customers, etc.
- Business Logic includes things like “What do I do after someone clicks 'checkout'?"
  - Answer: Show the “How will you pay for this?" screen, offering user's existing payment methods.
- Presentation Logic, which is run on Web Servers and in Web Browser clients, handles preparation and display of web pages to user.

Data Structures

- C connects to the database by structs of the following types:
  1. *Environments*: Represent the DBMS installation.
  2. *Connections*: Logins to the database.
  3. *Statements*: SQL statements to be passed to a connection.
  4. *Descriptions*: Records about tuples from a query, or parameters of a statement.

libpq

libpq is a native PostgreSQL library that allows C programs to interact with a PostgreSQL database.

- libpq is the basis for libraries supporting PostgreSQL application interfaces for other languages, including C++ and Python.

For libpq information, see:

- Chapter 33. libpq - C Library in PostgreSQL docs, particularly:
  - 31.1. Database Connection Control Functions
  - 31.3. Command Execution Functions
  - 33.21. libpq Example Programs
- PostgreSQL C tutorial from Zetcode

Compile your completed runShakespeareApplication.c by executing:

```
gcc -L/usr/include -lpq -o runShakespeareApplication runShakespeareApplication.c
```

Some Constants We Supply

```
/* These constants would normally be in a header file */

/* Maximum length of string used to submit a connection */
#define MAXCONNECTIONSTRINGSIZE 501

/* Maximum length of string used to submit a SQL statement */
#define MAXSQLSTATEMENTSTRINGSIZE 2001

/* Maximum length of string version of integer; you don't have to
use a value this big */
#define  MAXNUMBERSTRINGSIZE       20
```

Hope that you know how to convert:
- an integer to a string and
- a string to a integer in C.

Making a Connection

```
#include <stdio.h>
#include <stdlib.h>
#include <stderr.h>
#include "libpq-fe.h"

char conninfo[MAXCONNECTIONSTRINGSIZE];
 PGconn     *conn;
 PGresult   *res;

 PGconn *conn = PQconnectdb(conninfo);
```

libpq-fe.h contains definitions of enumerations, structures, constants, and functions of the C programming interface.

See /usr/include/libpq-fe.h on unix.ucsc.edu

conninfo is the parameter string for making a connection, providing:
host, dbname, userid, password

Your **dbname** is the same as your userid, so you can omit dbname.

conn is the connection to the DB

libpq documentation describe some security warnings. Security is vital … but which we won't discuss this topic.

Checking the Connection

```
PGconn *conn = PQconnectdb(conninfo);

 /* Check to see that the backend connection was successfully made */
   if (PQstatus(conn) != CONNECTION_OK)
   {
      fprintf(stderr, "Connection to database failed: %s",
            PQerrorMessage(conn));
      exit(EXIT_FAILURE);
   }
```

PQstatus checks status of connection.

PQerrorMessage returns latest error message associated with the connection.

Some Exit Functions

```
static void good_exit(PGconn *conn)
{
   PQfinish(conn);
   exit(EXIT_SUCCESS);
}
```

PQfinish closes connection to the server, and frees memory used by the PGconn object conn.

```
static void bad_exit(PGconn *conn)
{
   PQfinish(conn);
   exit(EXIT_FAILURE);
}
```

Only use these functions after a connection has successfully been established.

You'll lose credit if you don't free storage for connections and results in your programs when they're no longer needed!!!

Executing a SELECT Statement (1)

```
PGresult *res = PQexec(conn,
    "SELECT price, beer FROM Sells WHERE bar='Joe''s Bar' ");
```

You can execute any SQL statement (including BEGIN and COMMIT of a transaction) using PQexec.

```
if (PQresultStatus(res) != PGRES_TUPLES_OK)
{
   printf("No data retrieved\n");
   PQclear(res);
   good_exit(conn);
}
```

PGRES_TUPLES_OK indicates successful completion of a SELECT, even if 0 rows selected.

PQclear frees the storage associated with a PGresult.

```
/* Maybe we only want to print the first row */
printf("price is %s and beer is %s\n",
                    PQgetvalue(res, 0, 0),
                    PQgetvalue(res, 0, 1) );

PQclear(res);
```

PQgetvalue returns a string containing a single field value for one row of a PGresult.

Row and column numbers both **start at 0,** not at 1.

Executing a SELECT Statement (2)

Space between single quote and double quote isn't needed; it's just there for clarity.
No semicolon after SQL statement.

```
PGresult *res = PQexec(conn,
    "SELECT price, beer FROM Sells WHERE bar='Joe''s Bar' ");

if (PQresultStatus(res) != PGRES_TUPLES_OK)
    … see what's on the previous slide

/* Maybe we want to print all rows */
int n = PQntuples(res);
for (int j = 0; j < n; j++)
    printf("price is %s and beer is %s\n",
                PQgetvalue(res, j, 0),
                PQgetvalue(res, j, 1) );

PQclear(res);
```

PQntuples returns an integer, the number of tuples in the result.

PQgetvalue returns a string containing a single field value for one row of a PGresult.

Row and column numbers both **start at 0,** not at 1.

Executing an UPDATE Statement

```
PGresult *res = PQexec(conn,
     "UPDATE Sells SET price = price + 1.00 WHERE bar='Joe''s Bar' ");

if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
   printf("Update failed\n");
   PQclear(res);
   good_exit(conn);
}
fprintf("Number of tuples updated was %s\n",
                        PQcmdTuples(res));

PQclear(res);
```

PGRES_COMMAND_OK indicates successful completion of a command that can never return rows.

PQcmdTuples returns a <u>string</u> containing the number of rows affected by the SQL command,

Stored Functions and Procedures using libpq

- You call a Stored Procedure using a CALL statement, so you can execute a Stored Procedure using PQexec.

- You can use a Stored Function in a SQL statement anywhere that you can use a value of that type, so you can execute a SQL statement that uses a Stored Function using PQexec.
  - You can use
    ```
    SELECT myFunction(x, y);
    ```
    to execute Stored Function myFunction on parameters x and y.
  - The result is a single row that has a single attribute, whose type is the type that myFunction returns.
    - But remember that PQgetValue(res, 0, 0) returns a string, which may have to be converted to the correct type, e.g., by using the C function atoi
  - Lab4 uses a Stored Function.

SQL Injection is Dangerous!

You can execute <u>any</u> SQL statement (including BEGIN and COMMIT of a transaction) using PQexec.

```
PGresult *res = PQexec(conn,
    "SELECT price, beer FROM Sells WHERE bar='Joe''s Bar' ");
```

Actually, the string given to PQexec can contain <u>multiple SQL statements</u>, which are executed as a transaction (unless you've previously begun a transaction).

Beware!!! If you're constructing a SQL statement based on an input value provided by a user, that input might change the statement in ways that you had not anticipated!

```
char mycommand[MAXSQLSTATEMENTSTRINGSIZE] =
            “UPDATE Employees SET salary = salary + 100 WHERE name = ";

strcat (mycommand, user_input );
```

- What if user_input is “ 'Smith' OR department = 'Engineering' "?
- What if user_input is “ 'Smith'; DELETE FROM Employees"?

How Can You Prevent SQL Injection?

You can check that the types of user parameters match the types that are expected.

- But that won't help when the expected type is a string!
- (But you might check that the user input corresponds to an employee, for example.)

You can prevent SQL Injection by **separating the parameters from the SQL command**.

- Capabilities to do this exist in all database systems.

PostgreSQL has libpq functions that help prevent SQL Injection.

- PQexecParams
  - Give a command string and a list of specified parameter values, and the command is executed immediately, using the parameter values.
- PQprepare and PQexecPrepared
  - Give a command string and a list of parameter types to PQprepare, and you can execute that command repeatedly using different parameter values specified in PQexecPrepared.
- Parameters are referred to in the SQL command string as $1, $2, etc.

Use of these particular libpq functions is complex, so we're not going to describe details.

- In practice, you definitely should use such capabilities to prevent SQL Injection!
- **… but don't try to use these libpq functions in Lab4.**
- You won't be asked about them on the Final.

JDBC: A Very Rapid Summary

- The following slides on JDBC were used in CSE 180 when Lab4 required JDBC/Java, rather than libpq/C.

- We will go over these slides <u>extremely quickly</u>,
  - … but maybe you'll learn the gist of how JDBC works,
  - … even though you won't understand all the details,
  - … which is okay since you don't use JDBC in Lab4,
  - … and there will <u>not</u> be any JDBC questions on the CSE 180 Final.

JDBC

- *Java Database Connectivity* (JDBC) is a library similar to the SQL Call Level Interface standard and ODBC (for C), but with Java as the host language.

- For JDBC use with PostgreSQL, see:
  - Brief guide to using JDBC with PostgreSQL
  - Setting up JDBC Driver, including CLASSPATH ***
  - Information about queries and updates
  - Guide for defining stored procedures/functions

Making a Connection

The JDBC classes

```
import java.sql.*;

Connection myCon =
  DriverManager.getConnection(…);
```

URL of the database
your name, and password
go here.

Statements

- JDBC provides two classes:

1. *Statement* is an object that can accept a string that is a SQL statement and can execute such a string.
2. *PreparedStatement* is an object that has an associated SQL statement ready to execute.

Creating Statements

- The Connection class has methods to create Statements and PreparedStatements.

```
Statement stat1 = myCon.createStatement();

PreparedStatement stat2 =
    myCon.prepareStatement(
        "SELECT beer, price FROM Sells " +
        "WHERE bar = 'Joe' 's Bar' "
         );
```

Executing SQL Statements

- JDBC distinguishes queries from modifications, which it calls “updates."
- Statement and PreparedStatement each have methods executeQuery and executeUpdate.
  - For Statement: one argument: the query or modification to be executed.
  - For PreparedStatement: no argument.

Example: Update

- stat1 is a Statement.
- We can use it to insert a tuple:

```
stat1.executeUpdate(
  "INSERT INTO Sells " +
  "VALUES('Brass Rail','Bud',3.00)"
);
```

Example: Query

- stat2 is a PreparedStatement holding the query "SELECT beer, price FROM Sells WHERE bar = 'Joe''s Bar' ".

- executeQuery returns an object of class ResultSet; we'll examine that soon.

- The query:

```
ResultSet Menu = stat2.executeQuery();
```

Accessing the ResultSet

- An object of type ResultSet is a lot like a cursor.
- Method next() advances the “cursor" to the next tuple.
  - The first time next() is applied, it gets the first tuple.
  - If there are no more tuples, next() returns the value false.

Reminder of Example: Query

- stat2 is a PreparedStatement holding the query "SELECT beer, price FROM Sells WHERE bar = 'Joe''s Bar' ".

- executeQuery returns an object of class ResultSet; we'll examine that soon.

- The query:

```
ResultSet Menu = stat2.executeQuery();
```

Accessing Components of Tuples

- When a ResultSet refers to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet.
- Method get*X* (*i*), where *X* is some type, and *i* is the component number, returns the value of that component.
  - The value must have type *X*.

Example: Accessing Components

- Menu is the ResultSet for query "SELECT beer, price FROM Sells WHERE bar = 'Joe' 's Bar' ".
- Access beer and price from each tuple by:

```
while ( Menu.next() ) {
  theBeer = Menu.getString(1);
  thePrice = Menu.getFloat(2);
   /* do something with theBeer and
  thePrice */
}
```

JDBC Code: Try/Catch

```
public static void JDBCexample(String username, String passwd)
 {
    try (Connection mycon = DriverManager.getConnection(
           "jdbc:postgresql://cse180-db.lt.ucsc.edu/" + username, username, passwd);
          Statement stmt = mycon.createStatement();
         )
    {
           Do some actual work ....
    }
    catch (SQLException sqle) {
      System.out.println("SQLException : " + sqle);
    }
 }
```

Above syntax works with Java 7 and JDBC 4 onwards,
  so you can use it with our PostgresQL database and driver.

Resources opened in “try (….)" syntax (called “try with resources") are automatically closed at the end of the try block, but you should still explicitly close them.

Executing a Stored Function GoodBeers

Assume GoodBeers somehow finds all the good beers that are sold at a specific bar (theBar) that sell for under a particular price (thePrice).

- We won't tell you the secret of how GoodBeers procedure works.

```
PreparedStatement stmt = mycon.prepareStatement(
    "SELECT * FROM GoodBeers(?, ?)");
stmt.setString(1,theBar);        /* first parameter */
stmt.setFloat(2,thePrice);     /* second parameter */
 ResultSet result = stmt.executeQuery();
 while(result.next())    {
    theBeer= result.getString(1);
    /* do something with theBeer */
  }
```

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