

# **Real SQL**

# **Application Programming**

**Instructor: Shel Finkelstein**

*Reference:*

*A First Course in Database Systems,  
3<sup>rd</sup> edition, Chapter 9*

# Important Notices

- Midterm with Answers is on Piazza.
  - Grades were posted on Canvas on Monday, Nov 13.
    - Median was 83; no curve.
  - Exam was returned in class on Nov 13 and Nov 15.
    - Send email if you haven't received it yet, and I'll bring it to class.
- Lab3 assignment was posted on Sunday, Nov 5, and is due by Sunday, Nov 19, 11:59pm.
  - Lab3 has lots of parts (some hard), and is worth 13 points.
  - Please attend Labs to get help with Lab3.
- Lab4 assignment should be posted on Monday, Nov 20 or before, due on Sunday, Dec 3, 11:59pm.
  - Subject of Lab4 is Lecture 10 (Application Programming).
- Gradiance #4 will also be assigned before Thanksgiving.
  - No UCSC classes, Lab Sections, etc. on Thu Nov 23 and Fri Nov 24.

# 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 different!
  - Conventional programs written in C or Java, (or other languages) that interact with database using SQL.
  - Why?

# Approaches

1. Code in a specialized language is stored in the database itself (e.g., **Stored Procedure** languages such as PSM and PL/SQL).
2. SQL statements are **embedded in a host language** (e.g., C).
3. **Connection tools/libraries** are used to allow a **conventional language** to access a database (e.g., CLI, JDBC).

# Approach 1: Stored Procedures

- PSM, or “*persistent stored modules*,” allows us to store procedures as database schema elements.
- PSM = a mixture of conventional statements (if, while, etc.) and SQL.
- Lets us do things we cannot do in SQL alone.

# Basic PSM Form

```
CREATE PROCEDURE <name> (  
    <parameter list> )  
    <optional local declarations>  
    <body>;
```

# Basic PSM Form

```
CREATE FUNCTION <name> (  
    <parameter list> ) RETURNS <type>  
    <optional local declarations>  
    <body>;
```

# Parameters in PSM

- Unlike the usual name-type pairs in languages like C, PSM uses mode-name-type triples, where the *mode* can be:
  - IN = procedure uses value, does not change value.
  - OUT = procedure changes, does not use.
  - INOUT = both.
- Function parameters must be of mode IN. Functions returns value, but must have no side-effects on parameters.



## Example: Stored Procedure

- Let's write a procedure 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
```

Parameters are both  
read-only, not changed

**)**

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

The body ---  
a single insertion

# Invoking Procedures

- Use SQL/PSM statement CALL, with the name of the desired procedure and arguments.

- **Example:**

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

- Functions may be used in SQL expressions wherever a value of their return type is appropriate.

# Kinds of PSM statements – (1)

- RETURN <expression> sets the return value of a function.
  - Unlike C, etc., RETURN *does not* terminate function execution.
- DECLARE <name> <type> used to declare local variables.
- BEGIN . . . END for groups of statements.
  - Separate statements by semicolons.

# Kinds of PSM Statements – (2)

- **Assignment statements:**  
SET <variable> = <expression>;
  - Example: SET b = 'Bud' ;
- **Statement labels:** give a statement a label by prefixing a name and a colon.

# IF Statements

- Simplest form:  
IF <condition> THEN  
  <statements(s)>  
END IF;
- Add ELSE <statement(s)> if desired, as  
IF . . . THEN . . . ELSE . . . END IF;
- Add additional cases by ELSEIF <statements(s)>:  
IF ... THEN ... ELSEIF ... THEN ... ELSEIF ...  
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'.
  - $\geq 200$  customers: 'popular'.
- Function `Rate(b)` rates bar b.

# Example: IF (continued)

```
CREATE FUNCTION Rate (IN b CHAR(20) )
```

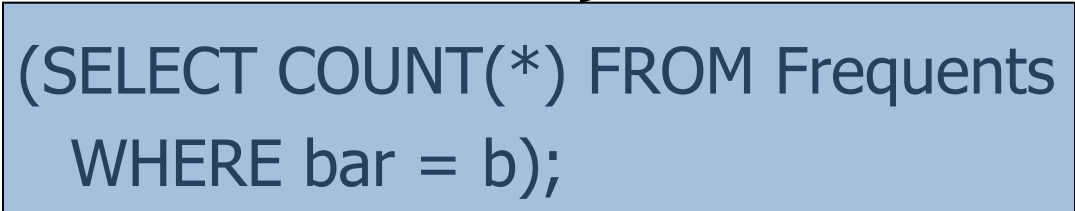
```
  RETURNS CHAR(10)
```

```
  DECLARE cust INTEGER;
```

```
  BEGIN
```

```
    SET cust = (SELECT COUNT(*) FROM Frequents  
                WHERE bar = b);
```

Number of  
customers of  
bar b



```
    IF cust < 100 THEN RETURN 'unpopular'  
    ELSEIF cust < 200 THEN RETURN 'average'  
    ELSE RETURN 'popular'  
    END IF;
```

Nested  
IF statement



```
  END;
```

Return occurs here, not at  
one of the RETURN statements





# Loops

- Basic form:  
    <loop name>: LOOP  
        <statements>  
    END LOOP;
- Exit from a loop by:  
    LEAVE <loop name>;

# Example: Exiting a Loop

```
loop1: LOOP
```

```
...
```

```
LEAVE loop1;  ← If this statement is executed ...
```

```
...
```

```
END LOOP;
```

← Control winds up here

# Other Loop Forms

- WHILE <condition>  
    DO <statements>  
    END WHILE;
- REPEAT <statements>  
    UNTIL <condition>  
    END REPEAT;

# Queries

- General SELECT-FROM-WHERE queries are *not* permitted in PSM.
- There are three ways to get the effect of a query:
  1. Queries producing one 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:

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

# SELECT . . . INTO ...

- Another way to get the value of a query that returns one tuple is by placing **INTO <variable>** after the SELECT clause.
- **Example:**

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

# 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>;

# Opening and Closing Cursors

- To use cursor  $c$ , we must issue the command:  
OPEN  $c$ ;
  - The query of  $c$  is evaluated, and  $c$  is set to point to the first tuple of the result.
- When finished with  $c$ , issue command:  
CLOSE  $c$ ;



# Fetching Tuples From a Cursor

- To get the next tuple from cursor *c*, issue command:

FETCH FROM *c* INTO *x*<sub>1</sub>, *x*<sub>2</sub>, ..., *x*<sub>*n*</sub> ;

- The *x* ' s are a list of variables, one for each component of the tuples referred to by *c*.
- *c* is moved automatically to the next tuple.

# Breaking Cursor Loops – (1)

- The usual way to use a cursor is to create a loop with a FETCH statement, and do something with each tuple fetched.
- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver.

# Breaking Cursor Loops – (2)

- Each SQL operation returns a *status*, which is a 5-digit character string.
  - For example:
    - '00000' means “Everything OK,”
    - '02000' means “Failed to find a tuple.”
- In PSM, we can get the value of the status in a variable called SQLSTATE.

# Breaking Cursor Loops – (3)

- We may declare a *condition*, which is a boolean variable that is true if and only if SQLSTATE has a particular value.
- **Example:** We can declare condition `NotFound` to represent 02000 by:

```
DECLARE NotFound CONDITION FOR  
        SQLSTATE '02000';
```

# Breaking Cursor Loops – (4)

- The structure of a cursor loop is thus:

```
cursorLoop: LOOP
```

```
...
```

```
  FETCH c INTO ... ;
```

```
  IF NotFound THEN LEAVE cursorLoop;
```

```
  END IF;
```

```
...
```

```
END LOOP;
```

## Example: Cursor

- Let's write a procedure that examines `Sells(bar, beer, price)`, and raises by one dollar the price of all beers at Joe's Bar that are under three dollars.
- Yes, we could write this as a simple UPDATE, but the details are instructive anyway.

# The Needed Declarations

```
CREATE PROCEDURE JoeGouge( )
```

```
    DECLARE theBeer CHAR(20);
```

```
    DECLARE thePrice REAL;
```

Used to hold  
beer-price pairs  
when fetching  
through cursor c

```
    DECLARE NotFound CONDITION FOR
```

```
        SQLSTATE ' 02000' ;
```

```
    DECLARE c CURSOR FOR
```

```
        (SELECT beer, price FROM Sells
```

```
            WHERE bar = ' Joe' ' s Bar' );
```

Returns Joe's menu

# The Procedure Body

```
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 bar = 'Joe' 's Bar' AND beer = theBeer;
    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.



# 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.

# PL/SQL

- 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)**
- IBM DB2: **SQL PL**
- MS SQL Server and Sybase: **Transact-SQL (T-SQL)**

# 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 some systems, the trigger body may include many of the kinds of statements that can be in a Stored Procedure.

# Approach 2: Embedded SQL

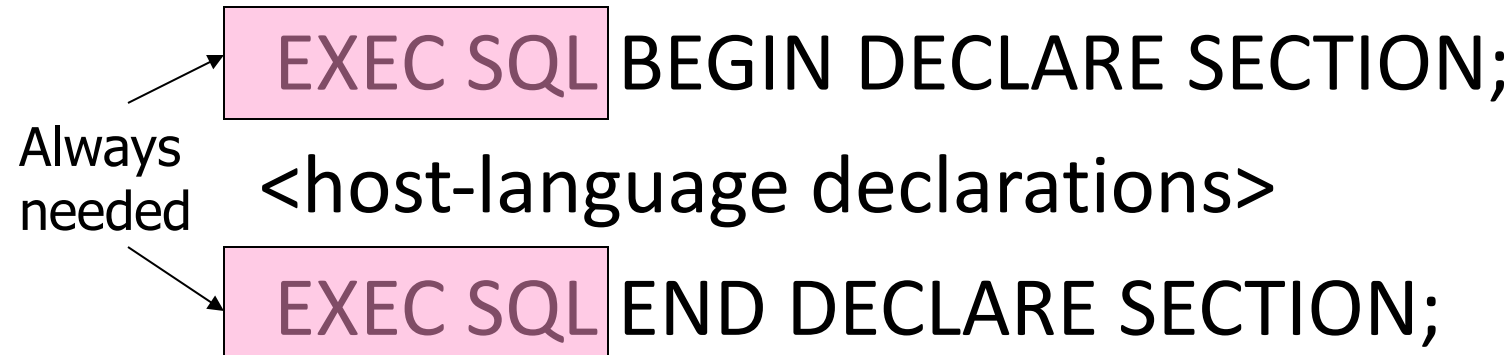
- **Key idea:** A pre-processor turns SQL statements into procedure calls that fit with the surrounding host-language code.
- All embedded SQL statements begin with EXEC SQL, so the pre-processor can find them easily.

# Shared Variables

- To connect SQL and the host-language program, the two parts must share some variables.
- Declarations of shared variables are bracketed by:

Always  
needed

```
EXEC SQL BEGIN DECLARE SECTION;  
<host-language declarations>  
EXEC SQL END DECLARE SECTION;
```



# Use of Shared Variables

- In SQL, the shared variables must be preceded by a colon.
  - They may be used as if they were constants provided by the host-language program.
  - They may get values from SQL statements and pass those values to the host-language program.
- In the host language, shared variables behave like any other variable.

# Example: Looking Up Prices

- We'll use C with embedded SQL to sketch the important parts of a function that obtains a beer and a bar, and looks up the price of that beer at that bar.
- Assumes database has the **Sells(bar, beer, price)** relation.

# Example: C with SQL

```
EXEC SQL BEGIN DECLARE SECTION;
```

```
char theBar[21], theBeer[21];
```

```
float thePrice;
```

```
EXEC SQL END DECLARE SECTION;
```

```
/* obtain values for theBar and theBeer */
```

```
EXEC SQL SELECT price INTO :thePrice
```

```
FROM Sells
```

```
WHERE bar = :theBar AND beer = :theBeer;
```

```
/* do something with thePrice */
```

Note 21-char  
arrays needed  
for 20 chars +  
endmarker

SELECT-INTO  
as in PSM



# Embedded Queries

- Embedded SQL has the same limitations as PSM regarding queries:
  - SELECT-INTO for a query guaranteed to produce a single tuple.
  - Otherwise, you have to use a cursor.
    - Small syntactic differences, but the key ideas are the same.

# Cursor Statements

- Declare a cursor *c* with:

```
EXEC SQL DECLARE c CURSOR FOR <query>;
```

- Open and close cursor *c* with:

```
EXEC SQL OPEN CURSOR c;
```

```
EXEC SQL CLOSE CURSOR c;
```

- Fetch from *c* by:

```
EXEC SQL FETCH c INTO <variable(s)>;
```

- You can write a macro NOT\_FOUND that is true if and only if the FETCH fails to find a tuple.
- If *c* is a cursor, you may use ... WHERE CURRENT OF *c*, just as in Stored Procedures.

# Example: Print Joe's Menu

- Let's write C + SQL to print Joe's menu – the list of beer-price pairs that we find in `Sells(bar, beer, price)` with `bar = Joe's Bar`.
- A cursor will visit each Sells tuple that has `bar = Joe's Bar`.

# Example: Declarations

```
EXEC SQL BEGIN DECLARE SECTION;  
    char theBeer[21]; float thePrice;  
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL DECLARE c CURSOR FOR  
    SELECT beer, price FROM Sells  
    WHERE bar = ' Joe' ' s Bar' ;
```



The cursor declaration goes outside the declare-section

# Example: Executable Part

```
EXEC SQL OPEN CURSOR c;
```

```
while(1) {
```

```
    EXEC SQL FETCH c
```

```
        INTO :theBeer, :thePrice;
```

```
    if (NOT_FOUND) break;
```

```
    /* format and print theBeer and thePrice */
```

```
}
```

```
EXEC SQL CLOSE CURSOR c;
```

The C style  
of breaking  
loops



# Need for Dynamic SQL

- Most applications use specific queries and modification statements to interact with the database.
  - The DBMS compiles EXEC SQL ... statements into specific procedure calls and produces an ordinary host-language program that uses a library.

# Dynamic SQL

- Preparing a query:

```
EXEC SQL PREPARE <query-name>  
FROM <text of the query>;
```

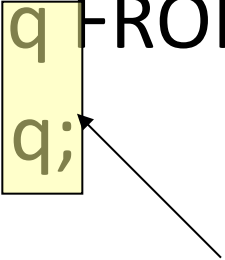
- Executing a query:

```
EXEC SQL EXECUTE <query-name>;
```

- “Prepare” means optimize query.
- Prepare once, Execute many times.

# Example: A Generic Interface

```
EXEC SQL BEGIN DECLARE SECTION;  
    char query[MAX_LENGTH];  
EXEC SQL END DECLARE SECTION;  
while(1) {  
    /* issue SQL> prompt */  
    /* read user's query into array query */  
    EXEC SQL PREPARE q FROM :query;  
    EXEC SQL EXECUTE q;  
}
```



q is an SQL "query variable" representing the optimized form of whatever statement is typed into :query



# Execute-Immediate

- If we are only going to execute the query once, we can combine the PREPARE and EXECUTE steps into one.

- Use:

```
EXEC SQL EXECUTE IMMEDIATE <text>;
```

## Example: Generic Interface Again

```
EXEC SQL BEGIN DECLARE SECTION;  
    char query[MAX_LENGTH];  
EXEC SQL END DECLARE SECTION;
```

```
while(1) {  
    /* issue SQL> prompt */  
    /* read user's query into array query */  
    EXEC SQL EXECUTE IMMEDIATE :query;  
}
```

# Approach 3: Host Language/SQL Interfaces via Libraries

- The third approach to connecting databases to conventional languages is to use library calls.
  1. C + CLI
  2. Java + JDBC
  3. PHP + PEAR/DB

# Three-Tier Architecture

- A common environment for using a database has three tiers of processors:
  1. *Web servers* --- talk to the user.
  2. *Application servers* --- execute business logic.
    - Often not used—logic in web tier
  3. *Database servers* --- get what the app servers (or web servers) need from the database.

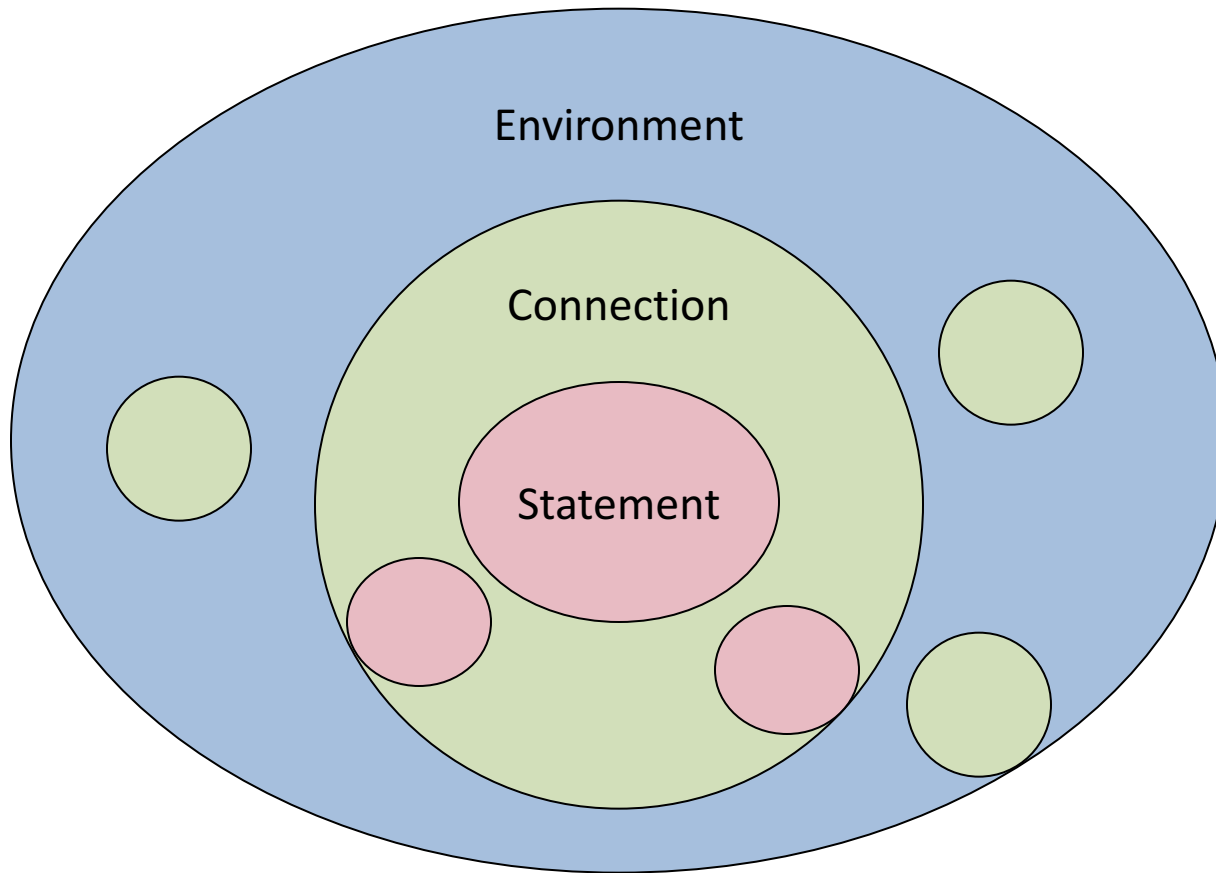
# Example: Amazon

- 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.
- Presentation layer, handled on web server and web browser, handles preparation and display of web pages

# Environments, Connections, Queries

- The database is, in many DB-access languages, an *environment*.
- Database servers maintain some number of *connections*, so app servers can connect to them and ask queries or perform modifications.
- The app server issues *statements* : queries and modifications, usually.

# Diagram to Remember



# SQL/CLI

- Instead of using a pre-processor (as in embedded SQL), we can use a library of functions.
  - The library for C is called SQL/CLI = “*Call-Level Interface*.”
  - Embedded SQL’s pre-processor will translate the EXEC SQL ... statements into CLI or similar calls, anyway.



# 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.

# JDBC

- *Java Database Connectivity* (JDBC) is a library similar to SQL/CLI, but with Java as the host language.
- Like CLI, but with a few differences.
- 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

```
import java.sql.*;  
Class.forName("com.mysql.jdbc.Driver");  
Connection myCon =  
    DriverManager.getConnection(...);
```

The JDBC classes

Loaded by  
forName

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

The driver  
for mySql;  
others exist

# 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 `getX ( 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 */  
}
```

# ExecuteQuery, ExecuteUpdate and Execute

- `executeQuery()`: Executes a SQL SELECT statement, and returns a `ResultSet` object.
- `executeUpdate()`: Executes a SQL UPDATE, INSERT or DELETE statement, and returns the number of affected rows.
  - May also be used with DDL, e.g., CREATE, DROP
- `execute()`: Executes either query or modification, and returns `TRUE` if query and `FALSE` if modification
  - `stat.getResultSet` for query result
  - `stat.getUpdateCount` for modification
- All methods may throw Exceptions

# Executing a Stored Procedure **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 */
}
```

# Executing a Stored Function

- Executing a Stored Function is similar to executing a Stored Procedure, except that what's returned is the result of the Stored Function.
  - Can be a scalar value (as in Lab4) or a table.
  - If it's a scalar value, that can be treated that as a table with one row.
- There is another way to execute Stored Procedures and Stored Functions, using the **CallableStatement** class (instead of Statement or PreparedStatement).
- This is not described well in the PostgreSQL documentation, but it's a good approach, and you may use it if you can figure it out.

# Approaches

## When/Why do you use each?

1. **Stored Procedure** languages such as PSM and PL/SQL)
2. SQL statements **embedded in a host language** (e.g., C)
3. **Connection tools/libraries** such as, CLI, and JDBC