CSE 180, Database Systems I
Shel Finkelstein, UC Santa Cruz
Lecture 8

Chapter 5: Advanced SQL
Sections 5.1 – 5.3

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Important Notices

- Lab3 solution has been posted on Piazza under Resources→Lab3.
- Lab4 assignment has been posted on Piazza under Resources→Lab4. See Piazza announcement about Lab4. Please read it soon!
  - Lab4 is due **Sunday, March 8**, by 11:59pm (2 week assignment).
  - Your solution should be submitted via Canvas as a zip file.
    - Late Lab Assignments will not be accepted.
    - Be sure that you submit the correct file!
  - Lab4 deals with material from this Lecture, Lecture 8.
    - Additional Piazza announcements describe some differences between PostgreSQL Stored Functions and the PSM standard.
  - Lab4 will be discussed at Lab Sections.
  - Load Data for Lab4 has now been posted; you'll need to use that Load Data to complete Lab4.

Important Notices

- Midterm been discussed in class and returned in class.
  - Please talk to to your TA first, and then to me if you disagree with grade.
- No Office Hour for me on Wed, Feb 26, 3:00-4:00pm; instead, my Office Hour this week will be Mon, Feb 24, 3:00-4:00pm in E2-249B.
  - Please come to my Office Hours to pick up Midterm if you haven't already picked it up
- Gradiance #4 will be posted this week.
- Attend Lectures, Lab Sections, Office Hours and LSS Tutoring.
  - See Piazza notices about LSS Tutoring with Jeshwanth Bheemanpally.

Outline

- Triggers
- Accessing SQL From a Programming Language
- Stored Functions and Procedures

Other Sections of Chapter 16 will be not covered in CSE 180.

- Recursive Queries
- Advanced Aggregation Features

Assertions

- These are database-schema elements, like relations or views.
- Defined by:

  CREATE ASSERTION <name>

  CHECK (<condition>);

- Condition may refer to any relation or attribute in the database schema.
- (**Not implemented** in most Relational DBMS because they're too complicated and expensive!)

Triggers: Motivation

- Assertions are powerful, but the DBMS often can't tell when they need to be checked …
  - … and they're probably not implemented by the DBMS.
- CHECK constraints are checked at known times, but they are not that powerful.
- Triggers let the user (often the DBA) decide when to check for any condition.

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
  - Triggers can be used to disambiguate updates of Views.
  - Can also be triggered by SELECT in some systems.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but were supported even earlier by most databases using non-standard syntax.
  - Systems still may have non-standard syntax.
  - Read the manuals for your DBMS.

©Silberschatz, Korth and Sudarshan

Event-Condition-Action Rules

- Another name for "trigger" is an *ECA Rule*, or *Event-Condition-Action* Rule
- *Event* : typically a type of database modification, e.g., "insert on Sells"
- *Condition* : Any SQL boolean-valued expression
- *Action* : Any SQL statements

©Silberschatz, Korth and Sudarshan

Triggering Events and Actions in SQL

- Triggering event can be **INSERT**, **DELETE** or **UPDATE**
- Triggers on update can be restricted to specific attributes
  - For example, **AFTER UPDATE OF** *takes* **ON** *grade*
- Values of attributes before and after an update can be referenced
  - **REFERENCING OLD ROW AS** : for deletes and updates
  - **REFERENCING NEW ROW AS** : for inserts and updates
- Triggers can be activated before an event. For example, convert blank grades to 'F'.

```
CREATE TRIGGER setF_trigger
BEFORE UPDATE OF takes
REFERENCING NEW ROW AS nrow
FOR EACH ROW
    WHEN (nrow.grade = ' ')
    BEGIN atomic
        SET nrow.grade = 'F';
    END;
```

Trigger to Maintain tot_cred value

- **CREATE TRIGGER** *credits_earned*
  AFTER UPDATE OF *takes* **ON** *grade*
  REFERENCING NEW ROW AS *nrow*
  REFERENCING OLD ROW AS *orow*
  FOR EACH ROW
  WHEN (*nrow.grade* <> 'F' **AND** *nrow.grade* **IS NOT NULL**)
  AND (*orow.grade* = 'F' **OR** *orow.grade* **IS NULL**)
  BEGIN atomic
  UPDATE *student*
  SET *tot_cred= tot_cred* +
  (**SELECT** *credits*
  FROM *course*
  WHERE *course.course_id = nrow.course_id*)
  WHERE *student.id = nrow.id*;
  END;

There are differences in Triggers in different DBMS. For example, Oracle omits **atomic**, and PostgreSQL limits trigger action to Stored Function/Procedure, as we'll discuss later.

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction.
  - Use **FOR EACH STATEMENT** instead of **FOR EACH ROW**
  - Use **REFERENCING OLD TABLE** or **REFERENCING NEW TABLE** to refer to temporary tables (called ***transition tables***) containing the affected rows.
  - Can be more efficient when dealing with SQL statements that update a large number of rows.

 ©Silberschatz, Korth and Sudarshan

Alternatives to Triggers

- Triggers were used earlier for tasks such as:
  - Maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are other ways of doing this now:
  - Some DBMS today provide built in materialized view facilities to maintain summary data.
    - More importantly, calculating aggregate data is now really fast in many DBMS.
  - Databases provide built-in support for replication.

Risks from Using Triggers

- Risk of unintended execution of triggers, for example, when:
  - Loading data from a backup copy
  - Replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

©Silberschatz, Korth and Sudarshan

Why Access SQL from a Programming Language?

A database programmer must have access to a general-purpose programming language for at least two reasons.

- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language.
- Non-declarative actions -- such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface -- cannot be done from within SQL.

Approaches to Accessing SQL from a Programming Language

There are two approaches to accessing SQL from a general-purpose programming language

- A general-purpose program -- can connect to and communicate with a database server using a collection of functions
- Embedded SQL -- provides a means by which a program can interact with a database server. The SQL statements are translated at compile time into function calls. At runtime, these function calls connect to the database using an API that provides dynamic SQL facilities.

A third approach, Stored Procedures/Functions, allows some programming language-like functionality to be executed within the database.

Stored Procedures and Functions

SQL Standard: PSM

- 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.
- Most DBMS, including PostgreSQL, implement Stored Procedures and Functions a little differently
  - We'll discuss some of the differences later in this Lectures, as well as on Piazza, since you'll need to know PostgreSQL's approach for Lab4.

©Silberschatz, Korth and Sudarshan

Basic PSM Form

CREATE PROCEDURE <name> (

<parameter list>)

<optional local declarations>

<body>;

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. A Function returns a value, but it must have no side-effects on parameters.

©Silberschatz, Korth and Sudarshan

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 the 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'.
  - >= 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);
        IF cust < 100 THEN RETURN ’ unpopular’
        ELSEIF cust < 200 THEN RETURN ’ average’
        ELSE RETURN ’ popular’
        END IF;
    END;
```

Number of customers of bar b

Nested IF statement

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>;

```
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

©Silberschatz, Korth and Sudarshan

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 x1, x2,…, x*n* ;

- 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.

©Silberschatz, Korth and Sudarshan

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;
    DECLARE NotFound CONDITION FOR
        SQLSTATE '02000';
    DECLARE c CURSOR FOR
        SELECT beer, price FROM Sells
        WHERE bar = 'Joe''s Bar';
```

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

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 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.

Note: There are some complex subtleties about when CURRENT OF Cursor can be used, so you probably don't want to use it in Lab4.

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)**
  - PostgreSQL only has Stored Functions, not Stored Procedures, but it extends Stored Functions in a non-standard way to do what Stored Procedures can do.
    - Similar to Oracle's PL/SQL.
  - We will post a Piazza note that gives info about PL/pgSQL 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 PostgreSQL, the trigger action **must** be a stored function
    - 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.

More on PL/pgSQL (for Lab4)

- PL/pgSQL has a bunch of differences from the PSM standard.
  - SQL standard is wonderful in many ways, but:
    - Standard doesn't specify everything.
    - Every system has extensions to standard.
    - Many systems deviate from some aspects of SQL standard.
- Some significant ways that PL/pgSQL differs from PSM are:
  1) No Stored Procedures in PL/pgSQL , but you can get equivalent by using non-standard extension of Stored Functions.
  2) Some slightly different syntax for Stored Functions in PL/pgSQL.
  3) Different way of handling "Not Found" for loops in PL/pgSQL.
- Lab4 was posted on Piazza on Monday, Feb 24, and there also are separate Piazza posts explaining 2) and 3), which you'll need in order to complete Lab4, which concerns Application Programming (this Lecture).
  - Lab4 is the one of the hardest Labs, so you should start it early!

Embedded SQL

Embedded SQL: Pre-processor

- 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.

©Silberschatz, Korth and Sudarshan

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.

©Silberschatz, Korth and Sudarshan

Example: Looking Up Prices

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

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 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, can 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;
}
```

JDBC

JDBC

- **JDBC** is a Java API for communicating with database systems supporting SQL.
- JDBC supports a variety of features for querying and updating data, and for retrieving query results.
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism (try/catch) to handle errors

©Silberschatz, Korth and Sudarshan

JDBC

- *Java Database Connectivity* (JDBC) is a library similar to ODBC for C, but with Java as the host language.
  - Like ODBC, but with a few differences.
  - JDBC Basics Tutorial

- 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 (Old Way)

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

The JDBC classes

Loaded by forName

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

The driver for mySql; others exist

Making a Connection (It's Easier Now)

```
import java.sql.*;
Connection myCon =
   DriverManager.getConnection( );
```

The JDBC classes

URL of your database, your name, and your 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
   */
}
```

SQL Injection

- Suppose query is constructed using
  - "select * from instructor where name = '" + name + "'"
  - What if name had quote in it, e.g., D'Souza?
- Worse yet, suppose the user, instead of entering a name, inputs:
  - X' or 'Y' = 'Y
- Then the resulting statement becomes:
  - "select * from instructor where name = '" + "X' or 'Y' = 'Y" + "'"
  - which is:
    - select * from instructor where name = 'X' or 'Y' = 'Y'
  - User could have even input the following:
    - X'; update instructor set salary = salary + 10000; --
- PreparedStatement internally prevents problems by using:
  "select * from instructor where name = 'X\' or \'Y\' = \'Y'
  - **Always use PreparedStatement, with user inputs as parameters, instead of concatenating user inputs into statement!**

JDBC Code: Try/Catch

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

NOTE: 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.

JDBC Code: Try/Catch (continued)

- "Update" to database, catching exception

```
try {
    stmt.executeUpdate(
        "insert into instructor values('77987', 'Kim', 'Physics', 98000)");
}
catch (SQLException sqle) {
   System.out.println("Could not insert tuple. " + sqle);
}
```

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

©Silberschatz, Korth and Sudarshan

Executing a Stored Function GoodBeers

Assume that 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 */
 }
```

When/Why Would 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 ODBC and JDBC

External Language Routines

- SQL allows us to define functions in a programming language such as Java, C#, C or C++.
  - Can sometimes be more efficient than functions defined in SQL.
  - Computations that cannot be carried out in SQL (or even withiin Stored Procedures) can be executed by these External routines.
- Declaring external language procedures and functions

```
create procedure dept_count_proc(in dept_name varchar(20),
                                 out count integer)
language C
external name   '/usr/avi/bin/dept_count_proc'

create function dept_count(dept_name varchar(20))
returns integer
language C
external name '/usr/avi/bin/dept_count'
```

External Language Routines (Cont.)

- Benefits of external language functions/procedures:
  - More efficient for many operations, and more expressive power.
- Drawbacks
  - Code to implement function may need to be loaded into database system and executed in the database system's address space.
    - Risk of accidental corruption of database structures
    - Security risks
  - There are alternatives, which give good security at the cost of potentially worse performance.
    - Execution in a separate process from the DB
    - Sandboxing

©Silberschatz, Korth and Sudarshan