Real SQL Programming

SQL in Real Programs

- We have seen only how SQL is used at the generic query interface – an environment where we sit at a terminal and ask queries of a database
- Reality is almost always different: conventional programs interacting with SQL

Options

1. Code in a specialized language is stored in the database itself (e.g., PSM, PL/pgsql)
2. SQL statements are embedded in a *host language* (e.g., C)
3. Connection tools are used to allow a conventional language to access a database (e.g., CLI, JDBC, PHP/DB)

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

Procedures in PostgreSQL

```
CREATE PROCEDURE <name>
  ([<arguments>]) AS $$
  <program>$$ LANGUAGE <lang>;
```

- PostgreSQL only supports functions:

```
CREATE FUNCTION <name>
  ([<arguments>]) RETURNS VOID AS $$
  <program>$$ LANGUAGE <lang>;
```

Parameters for Procedures

- Unlike the usual name-type pairs in languages like Java, procedures use mode-name-type triples, where the *mode* can be:
  - IN = function uses value, does not change
  - OUT = function changes, does not use
  - INOUT = both

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 = ’C.Ch.’, beer = *b*, and price = *p*
  - Used by Cafe Chino to add to their menu more easily

The Procedure

```
CREATE FUNCTION ChinoMenu (
   IN b        CHAR(20),
   IN p        REAL
) RETURNS VOID AS $$
INSERT INTO Sells
VALUES('C.Ch.', b, p);
$$ LANGUAGE plpgsql;
```

Parameters are both read-only, not changed

The body --- a single insertion

Invoking Procedures

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

```
CALL ChinoMenu('Eventyr', 50);
```

- Functions used in SQL expressions wherever a value of their return type is appropriate
- No CALL in PostgreSQL:

```
SELECT ChinoMenu('Eventyr', 50);
```

Kinds of PL/pgsql statements

- Return statement: RETURN <expression> returns value of a function
  - Like in Java, RETURN terminates the function execution
- Declare block: DECLARE <name> <type> used to declare local variables
- Groups of Statements: BEGIN . . . END
  - Separate statements by semicolons

Kinds of PL/pgsql statements

- Assignment statements:

  <variable> := <expression>;

  - Example: `b := 'Od.Cl.';`
- 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

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

Number of customers of bar b

Nested IF statement

Loops

- Basic form:

```
<<<label>>>
LOOP
    <statements>
END LOOP;
```

- Exit from a loop by:

```
EXIT <label> WHEN <condition>
```

Example: Exiting a Loop

```
<<loop1>> LOOP
    . . .
    EXIT loop1 WHEN ...;
    . . .
END LOOP;
```

If this statement is executed and the condition holds ...

... control winds up here

Other Loop Forms

- WHILE <condition> LOOP
```
    <statements>
END LOOP;
```
- Equivalent to the following LOOP:
```
LOOP
    EXIT WHEN NOT <condition>;
    <statements>
END LOOP;
```

Other Loop Forms

- FOR <name> IN <start> TO <end>

  ```
  LOOP
      <statements>
  END LOOP;
  ```

- Equivalent to the following block:

  ```
  <name> := <start>;
  LOOP  EXIT WHEN <name> > <end>;
      <statements>
      <name> := <name>+1;
  END LOOP;
  ```

Other Loop Forms

- FOR <name> IN REVERSE <start> TO <end> LOOP

```
    <statements>
END LOOP;
```

- Equivalent to the following block:

```
<name> := <start>;
LOOP  EXIT WHEN <name> < <end>;
    <statements>
    <name> := <name> - 1;
END LOOP;
```

Other Loop Forms

- FOR <name> IN <start> TO <end> BY <step> LOOP

```
<statements>
END LOOP;
```

- Equivalent to the following block:

```
<name> := <start>;
LOOP  EXIT WHEN <name> > <end>;
    <statements>
    <name> := <name>+<step>;
END LOOP;
```

Queries

- General SELECT-FROM-WHERE queries are *not* permitted in PL/pgsql
- 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 Cafe Chino charges for Odense Classic by:

```
p := (SELECT price FROM Sells
    WHERE bar = ’C.Ch’ AND
        beer = ’Od.Cl.’);
```

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 = 'C.Ch.' AND
        beer = 'Od.Cl.';
```

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_1, x_2,…,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)

- Many operations returns if a row has been found, changed, inserted, or deleted (SELECT INTO, UPDATE, INSERT, DELETE, FETCH)
- In plpgsql, we can get the value of the status in a variable called FOUND

Breaking Cursor Loops – (3)

- The structure of a cursor loop is thus:

```
<<cursorLoop>> LOOP
  …
  FETCH c INTO … ;
  IF NOT FOUND THEN EXIT cursorLoop;
  END IF;
  …
END LOOP;
```

Example: Cursor

- Let us write a procedure that examines Sells(bar, beer, price), and raises by 10 the price of all beers at Cafe Chino that are under 30
- Yes, we could write this as a simple UPDATE, but the details are instructive anyway

The Needed Declarations

```
CREATE FUNCTION RaisePrices()
   RETURNS VOID AS $$
   DECLARE   theBeer CHAR(20);
             thePrice REAL;
             c CURSOR FOR
  (SELECT beer, price FROM Sells
   WHERE bar = 'C.Ch.');
```

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

Returns Cafe Chino's price list

The Procedure Body

```
BEGIN
  OPEN c;
  <<menuLoop>> LOOP
    FETCH c INTO theBeer, thePrice;
    EXIT menuLoop WHEN NOT FOUND;
    IF thePrice < 30 THEN
        UPDATE Sells SET price = thePrice + 10
        WHERE bar = ’C.Ch.’ AND beer = theBeer;
    END IF;
  END LOOP;
  CLOSE c;
END;$$ LANGUAGE plpgsql;
```

Check if the recent FETCH failed to get a tuple

If Cafe Chino charges less than 30 for the beer, raise its price at at Cafe Chino by 10

Tuple-Valued Variables

- PL/pgsql allows a variable *x* to have a tuple type
- x R%ROWTYPE gives *x* the type of R’s tuples
- *R* could be either a relation or a cursor
- x.a gives the value of the component for attribute *a* in the tuple *x*

Example: Tuple Type

- Repeat of RaisePrices() declarations with variable *bp* of type beer-price pairs

```
CREATE FUNCTION RaisePrices()
  RETURNS VOID AS $$
  DECLARE    CURSOR c IS
  SELECT beer, price FROM Sells
  WHERE bar = ’C.Ch.’;
             bp c%ROWTYPE;
```

RaisePrices() Body Using *bp*

```
BEGIN
   OPEN c;
   LOOP
     FETCH c INTO bp;
     EXIT WHEN NOT FOUND;
     IF bp.price < 30 THEN
       UPDATE Sells SET price = bp.price + 10
       WHERE bar = ’C.Ch.’ AND beer = bp.beer;
     END IF;
   END LOOP;
   CLOSE c;
END;
```

Components of bp are obtained with a dot and the attribute name

Database-Connection Libraries

Host/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 the business logic
  3. *Database servers* – get what the app 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

Environments, Connections, Queries

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

JDBC

- *Java Database Connectivity* (JDBC) is a library similar for accessing a DBMS with Java as the host language
- 221 drivers available: PostgreSQL, MySQL, Oracle, ODBC, ...
- `http://jdbc.postgresql.org/`

Making a Connection

The JDBC classes

```
import java.sql.*;
...
Class.forName(“org.postgresql.Driver”);
Connection myCon =
  DriverManager.getConnection(…);
...
```

Loaded by forName

URL of the database your name, and password go here

The driver for postgresql; others exist

URL for PostgreSQL database

- jdbc:postgresql://<host>[:<port>]/<database>?user=<user>&password=<password>
- Alternatively use getConnection variant:
- getConnection("jdbc:postgresql://<host>[:<port>]/<database>", <user>, <password>);
- DriverManager.getConnection("jdbc:postgresql://10.110.4.210/petersk09", "petersk09", "geheim");

Statements

- JDBC provides two classes:
  1. *Statement* = an object that can accept a string that is a SQL statement and can execute such a string
  2. *PreparedStatement* = 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.createStatement(
     ”SELECT beer, price FROM Sells ” +
     ”WHERE bar = ’C.Ch.’ ”
   );
```

createStatement with no argument returns a Statement; with one argument it returns a PreparedStatement

Executing SQL Statements

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

Example: Update

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

```
stat1.executeUpdate(
  ”INSERT INTO Sells ” +
  ”VALUES(’C.Ch.’,’Eventyr’,30)”
);
```

Example: Query

- stat2 is a PreparedStatement holding the query ”SELECT beer, price FROM Sells WHERE bar = ’C.Ch.’ ”
- executeQuery returns an object of class ResultSet – we’ll examine it later
- The query:

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

Accessing the ResultSet

- An object of type ResultSet is something 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

Accessing Components of Tuples

- When a ResultSet is referring 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 = ResultSet for query “SELECT beer, price FROM Sells WHERE bar = ’C.Ch.’ ”
- Access beer and price from each tuple by:

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

Important Details

- Reusing a Statement object results in the ResultSet being closed
  - Always create new Statement objects using createStatement() or explicitly close ResultSets using the close method
- For transactions, for the Connection con use con.setAutoCommit(false) and explicitly con.commit() or con.rollback()
  - If AutoCommit is false and there is no commit, closing the connection = rollback

PHP

- A language to be used for actions within HTML text
- Indicated by <?PHP code ?>.
- DB library exists within *PEAR* (PHP Extension and Application Repository)
  - Include with `include(DB.php)`

Variables in PHP

- Must begin with $
- OK not to declare a type for a variable
- But you give a variable a value that belongs to a “class,” in which case, methods of that class are available to it

String Values

- PHP solves a very important problem for languages that commonly construct strings as values:
  - How do I tell whether a substring needs to be interpreted as a variable and replaced by its value?
- PHP solution: Double quotes means replace; single quotes means do not

Example: Replace or Not?

```
$100 = "one hundred dollars";
$Peter = 'You owe me $100.';
$Lars = "You owe me $100.";
```

- Value of $Peter is 'You owe me $100', while the value of $Lars is 'You owe me one hundred dollars'

PHP Arrays

- Two kinds: *numeric* and *associative*
- Numeric arrays are ordinary, indexed 0,1,…
  - Example: $a = array(”Paul”, ”George”, ”John”, ”Ringo”);
    - Then $a[0] is ”Paul”, $a[1] is ”George”, and so on

Associative Arrays

- Elements of an associative array $a are pairs x => y, where *x* is a key string and *y* is any value
- If x => y is an element of $a, then $a[x] is *y*

Example: Associative Arrays

- An environment can be expressed as an associative array, e.g.:

```
$myEnv = array(
  ”phptype”   => ”pgsql”,
  ”hostspec” => ”localhost”,
  ”port”       => ”5432”,
  ”database” => ”petersk09”,
  ”username” => ”petersk09”,
  ”password” => ”geheim”);
```

Making a Connection

- With the DB library imported and the array $myEnv available:

```
$myCon = DB::connect($myEnv);
```

Function connect in the DB library

Class is Connection because it is returned by DB::connect()

Executing SQL Statements

- Method query applies to a Connection object
- It takes a string argument and returns a result
  - Could be an error code or the relation returned by a query

Example: Executing a Query

- Find all the bars that sell a beer given by the variable $beer

Method application

Concatenation in PHP

```
$beer = 'Od.Cl.';
$result = $myCon->query(
  "SELECT bar FROM Sells" .
  "WHERE beer = '$beer';");
```

Remember this variable is replaced by its value.

Cursors in PHP

- The result of a query *is* the tuples returned
- Method fetchRow applies to the result and returns the next tuple, or FALSE if there is none

Example: Cursors

```
while ($bar = $result->fetchRow())
{
  // do something with $bar
}
```

Example: Tuple Cursors

```
$bar = "C.Ch.";
$menu = $myCon->query(
  "SELECT beer, price FROM Sells
  WHERE bar = '$bar';");
while ($bp = $result->fetchRow())
{
  print $bp[0] . " for " . $bp[1];
}
```