Real SQL Programming

Persistent Stored Modules (PSM)
PL/SQL
Embedded SQL

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/SQL).
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.
Basic PSM Form

CREATE PROCEDURE <name> (
   <optional local declarations>
)

<body>

Function alternative:

CREATE FUNCTION <name> (
   <parameter list>
) RETURNS <type>

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.
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

```sql
CREATE PROCEDURE JoeMenu (  
  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:

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

- Functions 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`>
  - END IF;
- **Add additional cases by ELSEIF**
  - IF `<condition>` THEN `<statements`>
  - END IF;
  - ELSEIF `<condition>` THEN `<statements`>
  - END IF;
  - ... ELSEIF `<condition>` THEN `<statements`>
  - END IF;
  - ELSE `<statements`>
  - END IF;
Example: IF

Let’s rate bars by how many customers they have, based on \texttt{Frequents(drinker,bar)}.
- <100 customers: ‘unpopular’.
- 100-199 customers: ‘average’.
- >= 200 customers: ‘popular’.

Function \texttt{Rate(b)} rates bar b.

---

Example: IF (continued)

CREATE FUNCTION Rate (IN b CHAR(20) )
RETURNS CHAR(10)
DECLARE cust INTEGER;
BEGIN
SET cust =

Number of customers of bar b

Return occurs here, not at one of the RETURN statements

Nested IF statement
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 \texttt{Sells(bar, beer, price)}, we can get the price Joe charges for Bud by:

\[
\text{SET } p = \text{ (SELECT price FROM Sells WHERE bar = 'Joe''s Bar' AND beer = 'Bud');}
\]

SELECT . . . INTO

◆ Another way to get the value of a query that returns one tuple is by placing \texttt{INTO <variable>} after the \texttt{SELECT} clause.

◆ Example:

\[
\text{SELECT price INTO p FROM Sells WHERE bar = 'Joe''s Bar' AND beer = 'Bud';}
\]
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,...,xn ;
  
◆ 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 = “Everything OK,” and 02000 = “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:

```sql
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 $1 the price of all beers at Joe’s Bar that are under $3.

- Yes, we could write this as a simple `UPDATE`, but the details are instructive anyway.
The Needed Declarations

CREATE PROCEDURE JoeGouge()

DECLARE NotFound CONDITION FOR SQLSTATE '02000';
DECLARE c CURSOR FOR

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;
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 a variant of SQL/PSM which it calls PL/SQL.
- PL/SQL not only allows you to create and store procedures or functions, but it can be run from the *generic query interface* (sqlplus), like any SQL statement.
- Triggers are a part of PL/SQL.

Trigger Differences

- Compared with SQL standard triggers, Oracle has the following differences:
  1. Action is a PL/SQL statement.
  2. New/old tuples referenced automatically.
  3. Strong constraints on trigger actions designed to make certain you can’t fire off an infinite sequence of triggers.
- See on-line or-triggers.html document.
SQLPlus

- In addition to stored procedures, one can write a PL/SQL statement that looks like the body of a procedure, but is executed once, like any SQL statement typed to the generic interface.
  - Oracle calls the generic interface “sqlplus.”
  - PL/SQL is really the “plus.”

Form of PL/SQL Statements

DECLARE
  <declarations>
BEGIN
  <statements>
END;
.un
- The DECLARE section is optional.
Form of PL/SQL Procedure

CREATE OR REPLACE PROCEDURE
<name> (<arguments>)
<optional declarations>
BEGIN
<PL/SQL statements>
END;

PL/SQL Declarations and Assignments

◆ The word DECLARE does not appear in front of each local declaration.
  ♦ Just use the variable name and its type.
◆ There is no word SET in assignments, and := is used in place of =.
  ♦ Example: x := y;
PL/SQL Procedure Parameters

- There are several differences in the forms of PL/SQL argument or local-variable declarations, compared with the SQL/PSM standard:
  1. Order is name-mode-type, not mode-name-type.
  2. INOUT is replaced by IN OUT in PL/SQL.
  3. Several new types.

PL/SQL Types

- In addition to the SQL types, NUMBER can be used to mean INT or REAL, as appropriate.
- You can refer to the type of attribute $x$ of relation $R$ by $R.x\%\text{TYPE}$.
  - Useful to avoid type mismatches.
  - Also, $R\%\text{ROWTYPE}$ is a tuple whose components have the types of $R$’s attributes.
Example: JoeMenu

- Recall the procedure `JoeMenu(b,p)` that adds beer \( b \) at price \( p \) to the beers sold by Joe (in relation Sells).
- Here is the PL/SQL version.

Procedure JoeMenu in PL/SQL

```sql
CREATE OR REPLACE PROCEDURE JoeMenu ( 
    b IN -- Notice these types 
    p IN -- will be suitable 
) AS 
BEGIN 
    INSERT INTO Sells 
    VALUES (‘Joe’s Bar’, b, p); 
END; 
run
```
PL/SQL Branching Statements

- Like IF ... in SQL/PSM, but:
- Use ELSIF in place of ELSEIF.
- Viz.: IF ... THEN ... ELSEIF ... THEN ... ELSEIF ... THEN ... END IF;

PL/SQL Loops

- LOOP ... END LOOP as in SQL/PSM.
- Instead of LEAVE ... , PL/SQL uses EXIT WHEN <condition>.
- And when the condition is that cursor c has found no tuple, we can write c%NOTFOUND as the condition.
PL/SQL Cursors

◆ The form of a PL/SQL cursor declaration is:
  CURSOR <name> IS <query>;
◆ To fetch from cursor c, say:
  FETCH c INTO <variable(s)>;

Example: JoeGouge() in PL/SQL

◆ Recall JoeGouge() sends a cursor through the Joe’s-Bar portion of Sells, and raises by $1 the price of each beer Joe’s Bar sells, if that price was initially under $3.
**Example: JoeGouge() Declarations**

CREATE OR REPLACE PROCEDURE JoeGouge() AS
theBeer Sells.beer%TYPE;
thePrice Sells.price%TYPE;
CURSOR c IS
SELECT beer, price FROM Sells
WHERE bar = 'Joe''s Bar';

**Example: JoeGouge() Body**

BEGIN
OPEN c;
LOOP
  FETCH c INTO theBeer, thePrice;
  IF thePrice < 3.00 THEN
    UPDATE Sells
    WHERE bar = 'Joe''s Bar' AND beer = theBeer;
  END IF;
END LOOP;
CLOSE c;
END;
Tuple-Valued Variables

- PL/SQL 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 JoeGouge() declarations with variable $bp$ of type beer-price pairs.

```sql
CREATE OR REPLACE PROCEDURE JoeGouge() AS
    CURSOR c IS
    SELECT beer, price FROM Sells
    WHERE bar = 'Joe''s Bar';
    bp c%ROWTYPE;
```
JoeGouge() Body Using $bp$

BEGIN
OPEN c;
LOOP
  FETCH c INTO $bp$;
  EXIT WHEN c%NOTFOUND;
  IF $bp$ < 3.00 THEN
    UPDATE Sells SET price = $bp$ + 1.00
    WHERE bar = 'Joe''s Bar' AND beer = $bp$;
  END IF;
END LOOP;
CLOSE c;
END;

Embedded SQL

◆ **Key idea:** A preprocessor turns SQL statements into procedure calls that fit with the surrounding host-language code.

◆ All embedded SQL statements begin with EXEC SQL, so the preprocessor 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:

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

Use of Shared Variables

◆ In SQL, the shared variables must be preceded by a colon.
  ▪ They may be used as 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 our usual `Sells(bar, beer, price)` relation.

**Example: C Plus SQL**

```sql
EXEC SQL BEGIN DECLARE SECTION;
char bar[20];
float thePrice;
EXEC SQL END DECLARE SECTION;
/* obtain values for theBar and theBeer */
EXEC SQL SELECT INTO thePrice FROM Prices WHERE bar = bar; /* 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 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)>;
  - Macro NOT FOUND is true if and only if the FETCH fails to find a tuple.
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

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

The cursor declaration goes outside the declare-section
Example: Executable Part

EXEC SQL OPEN CURSOR c;
{
  EXEC SQL FETCH c INTO :theBeer, :thePrice;
  /* format and print theBeer and thePrice */
}
EXEC SQL CLOSE CURSOR c;

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.
◆ What about sqlplus, which doesn’t know what it needs to do until it runs?
Dynamic SQL

◆ Preparing a query:
EXEC SQL PREPARE <query-name>
   FROM <text of the query>;
◆ Executing a query:
EXEC SQL EXECUTE <query-name>;
◆ “Prepare” = 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 FROM :query;
   EXEC SQL EXECUTE q

q is an SQL 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;
}