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).

Control-of-Flow Language Elements

- Statement Level
  - BEGIN … END block
  - IF … ELSE block
  - WHILE constructs
- Row Level
  - CASE expression

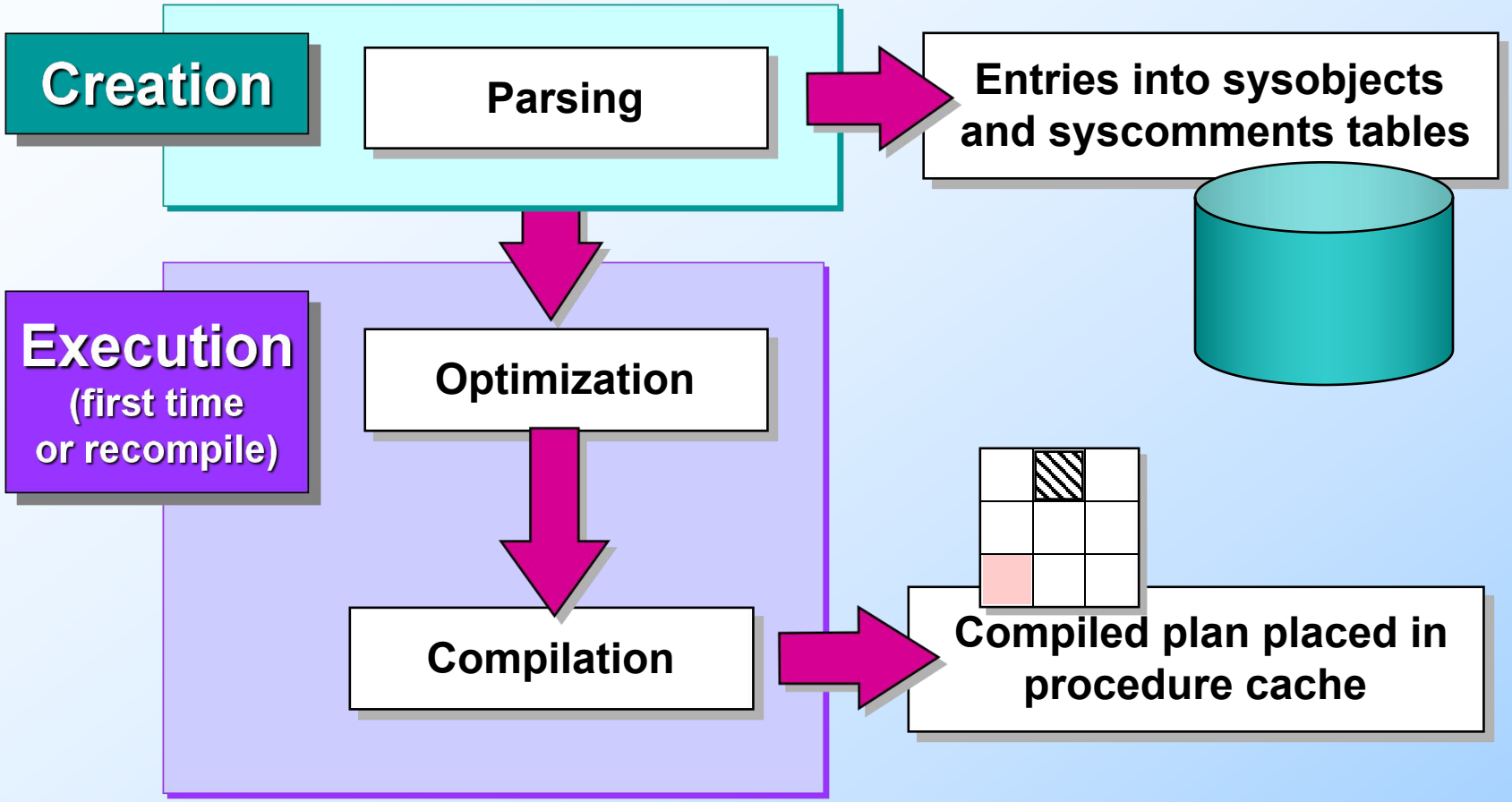
Example

```
DECLARE @n tinyint
SET @n = 5
IF (@n BETWEEN 4 and 6)
 BEGIN
  WHILE (@n > 0)
   BEGIN
    SELECT  @n AS 'Number'
      ,CASE
        WHEN (@n % 2) = 1
          THEN 'EVEN'
        ELSE 'ODD'
       END AS 'Type'
    SET @n = @n - 1
   END
 END
ELSE
 PRINT 'NO ANALYSIS'
```

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.

Initial Processing of Stored Procedures



Advantages of Stored Procedures

- Share Application Logic
- Shield Database Schema Details
- Provide Security Mechanisms
- Improve Performance
- Reduce Network Traffic

Basic PSM Form

CREATE PROCEDURE <name> (
    <parameter list>)
  <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

```
CREATE PROCEDURE JoeMenu (
    IN b    CHAR(20),
    IN p    REAL
)
INSERT INTO Sells
VALUES('Joe"s Bar', b, p);
```

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 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(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>

Example: Exiting a Loop

```
loop1: LOOP
        . . .
        LEAVE loop1;
        . . .
END LOOP;
```

If this statement is executed . . .

Control winds up here

Other Loop Forms

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

Executing Stored Procedures

- Executing a Stored Procedure by Itself

```
EXEC OverdueOrders
```

- Executing a Stored Procedure Within an INSERT Statement

```
INSERT INTO Customers
EXEC EmployeeCustomer
```

Using Input Parameters

- Validate All Incoming Parameter Values First
- Provide Appropriate Default Values and Include Null Checks

```
CREATE PROCEDURE dbo.[Year to Year Sales]
  @BeginningDate DateTime, @EndingDate DateTime
AS
IF @BeginningDate IS NULL OR @EndingDate IS NULL
BEGIN
   RAISERROR('NULL values are not allowed', 14, 1)
   RETURN
END
SELECT O.ShippedDate,
       O.OrderID,
       OS.Subtotal,
       DATENAME(yy,ShippedDate) AS Year
FROM ORDERS O INNER JOIN [Order Subtotals] OS
   ON O.OrderID = OS.OrderID
WHERE O.ShippedDate BETWEEN @BeginningDate AND @EndingDate
```

Executing Stored Procedures Using Input Parameters

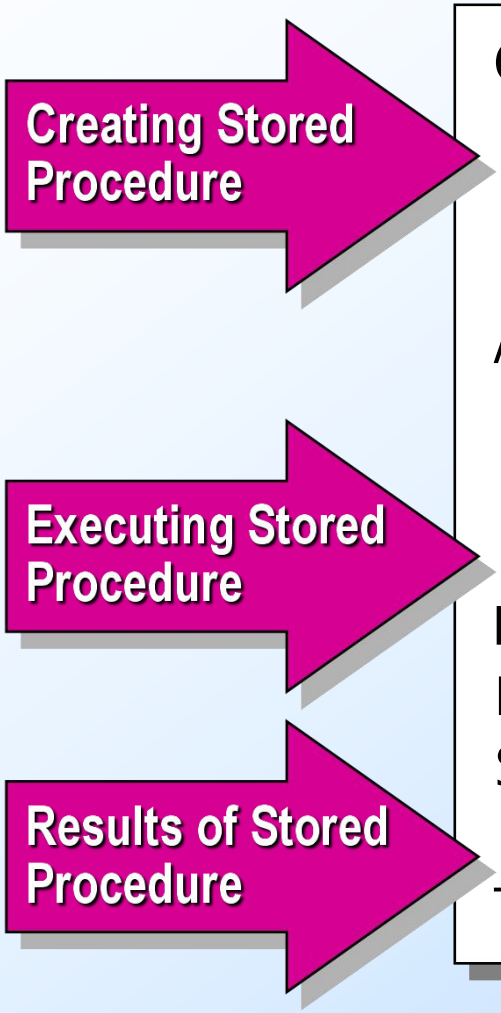
- Passing Values by Parameter Name

```
EXEC AddCustomer
    @CustomerID = 'ALFKI',
    @ContactName = 'Maria Anders',
    @CompanyName = 'Alfreds Futterkiste',
    @ContactTitle = 'Sales Representative',
    @Address = 'Obere Str. 57',
    @City = 'Berlin',
    @PostalCode = '12209',
    @Country = 'Germany',
    @Phone = '030-0074321'
```

- Passing Values by Position

```
EXEC AddCustomer 'ALFKI2', 'Alfreds
Futterkiste', 'Maria Anders', 'Sales
Representative', 'Obere Str. 57', 'Berlin',
NULL, '12209', 'Germany', '030-0074321'
```

Returning Values Using Output Parameters




```
CREATE PROCEDURE dbo.MathTutor
    @m1 smallint,
    @m2 smallint,
    @result smallint OUTPUT
AS
    SET @result = @m1* @m2
```

```
DECLARE @answer smallint
EXECUTE MathTutor 5,6, @answer OUTPUT
SELECT 'The result is: ', @answer
```

```
The result is:  30
```

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 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 = "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:

```
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 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.

游标的运行

```
EXEC SQL BEGIN    ……   //定义主变量
EXEC SQL DECLARE tnames_cursor CURSOR
   FOR SELECT name,xh FROM xjb
      WHERE xb='男'
      ORDER BY xh;
EXEC SQL OPEN tnames_cursor;
EXEC SQL FETCH tnames_cursor INTO :PNAME,:PXH;
WHILE (:PXH!=NULL)
   {
   printf("%s,%s\n",PNAME,PXH);
      EXEC SQL FETCH tnames_cursor INTO :PNAME,:PXH
   }
EXEC SQL CLOSE tnames_cursor;
EXEC SQL DEALLOCATE tnames_cursor;
```

:PNAME	:PXH
张三	003
李四	004
王五	005
孙六	006
陈七	007
曹八	008
.	
.	
.	

Results

张三	003
李四	004
王五	005
孙六	006
陈七	007
曹八	008
...	

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.

What Is a User-defined Function?

- Scalar Functions
  - Similar to a built-in function
- Multi-Statement Table-valued Functions
  - Content like a stored procedure
  - Referenced like a view
- In-Line Table-valued Functions
  - Similar to a view with parameters
  - Returns a table as the result of single SELECT statement

Defining User-defined Functions

- Creating a User-defined Function
- Creating a Function with Schema Binding
- Setting Permissions for User-defined Functions
- Altering and Dropping User-defined Functions

Creating a User-defined Function

- **Creating a Function**

```
USE Northwind
CREATE FUNCTION fn_NewRegion
  (@myinput nvarchar(30))
  RETURNS nvarchar(30)
BEGIN
  IF @myinput IS NULL
  SET @myinput = 'Not Applicable'
  RETURN @myinput
END
```

- **Restrictions on Functions**

Creating a Function with Schema Binding

- Referenced User-defined Functions and Views Are Also Schema Bound
- Objects Are Not Referenced with a Two-Part Name
- Function and Objects Are All in the Same Database
- Have Reference Permission on Required Objects

Setting Permissions for User-defined Functions

- Need CREATE FUNCTION Permission
- Need EXECUTE Permission
- Need REFERENCE Permission on Cited Tables, Views, or Functions
- Must Own the Function to Use in CREATE or ALTER TABLE Statement

Altering and Dropping User-defined Functions

- Altering Functions

```
ALTER FUNCTION dbo.fn_NewRegion
   <New function content>
```

  - Retains assigned permissions
  - Causes the new function definition to replace existing definition

- Dropping Functions

```
DROP FUNCTION dbo.fn_NewRegion
```

Examples of User-defined Functions

- Using a Scalar User-defined Function
- Example of a Scalar User-defined Function
- Using a Multi-Statement Table-valued Function
- Example of a Multi-Statement Table-valued Function
- Using an In-Line Table-valued Function
- Example of an In-Line Table-valued Function

Using a Scalar User-defined Function

- RETURNS Clause Specifies Data Type
- Function Is Defined Within a BEGIN and END Block
- Return Type Is Any Data Type Except text, ntext, image, cursor, or timestamp

Example of a Scalar User-defined Function

- **Creating the Function**

```
CREATE FUNCTION fn_DateFormat
      (@indate datetime, @separator char(1))
   RETURNS Nchar(20)
   AS
   BEGIN
      RETURN
      CONVERT(Nvarchar(20), datepart(mm,@indate))
      + @separator
      + CONVERT(Nvarchar(20), datepart(dd, @indate))
      + @separator
      + CONVERT(Nvarchar(20), datepart(yy, @indate))
   END
```

- **Calling the Function**

```
SELECT dbo.fn_DateFormat(GETDATE(), ':')
```

Using a Multi-Statement Table-valued Function

- BEGIN and END Enclose Multiple Statements
- RETURNS Clause Specifies table Data Type
- RETURNS Clause Names and Defines the Table

Example of a Multi-Statement Table-valued Function

- Creating the Function

```
CREATE FUNCTION fn_Employees (@length nvarchar(9))
RETURNS @fn_Employees table
   (EmployeeID int PRIMARY KEY NOT NULL,
   [Employee Name] nvarchar(61) NOT NULL)
AS
BEGIN
   IF @length = 'ShortName'
      INSERT @fn_Employees SELECT EmployeeID, LastName
      FROM Employees
   ELSE IF @length = 'LongName'
      INSERT @fn_Employees SELECT EmployeeID,
      (FirstName + ' ' + LastName) FROM Employees
RETURN
END
```

- Calling the Function

```
SELECT * FROM dbo.fn_Employees('LongName')
Or
SELECT * FROM dbo.fn_Employees('ShortName')
```

Using an In-Line Table-valued Function

- Content of the Function Is a SELECT Statement
- Do Not Use BEGIN and END
- RETURN Specifies table as the Data Type
- Format Is Defined by the Result Set

Example of an In-Line Table-valued Function

- **Creating the Function**

```
CREATE FUNCTION fn_CustomerNamesInRegion
   ( @RegionParameter nvarchar(30) )
RETURNS table
AS
RETURN (
   SELECT CustomerID, CompanyName
   FROM Northwind.dbo.Customers
   WHERE Region = @RegionParameter
   )
```

- **Calling the Function Using a Parameter**

```
SELECT * FROM fn_CustomerNamesInRegion(N'WA')
```

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;
.
run
```

- The DECLARE section is optional.

Form of PL/SQL Procedure

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

Notice AS needed here

Needed to store procedure in database; does not really run it.

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%TYPE.
  - Useful to avoid type mismatches.
  - Also, R%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

```
CREATE OR REPLACE PROCEDURE JoeMenu (
   b IN Sells.beer%TYPE,
   p IN Sells.price%TYPE
) AS
   BEGIN
        INSERT INTO Sells
        VALUES ('Joe''s Bar', b, p);
   END;
.
run
```

Notice these types will be suitable for the intended uses of *b* and *p*.

PL/SQL Branching Statements

- Like IF … in SQL/PSM, but:
- Use ELSIF in place of ELSEIF.
- Viz.: IF … THEN … ELSIF … THEN … ELSIF … THEN … ELSE … 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;
      EXIT WHEN c%NOTFOUND;
      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;
```

How PL/SQL breaks a cursor loop

Note this is a SET clause in an UPDATE, not an assignment. PL/SQL uses := for assignments.

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.

```
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.price < 3.00 THEN
            UPDATE Sells SET price = bp.price + 1.00
            WHERE bar = 'Joe"s Bar' AND beer = bp.beer;
        END IF;
    END LOOP;
    CLOSE c;
END;
```

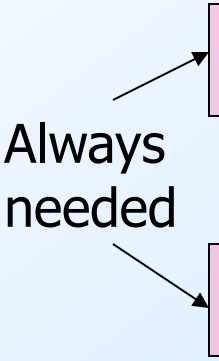
Components of bp are obtained with a dot and the attribute name

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:




```
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 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

```
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)>;

  - 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

```
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.
- 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 q 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;
}
```