Chapter 9 SQL in a server environment

- SQL in a Programming Environment

    embedded SQL

    persistent stored modules

- Database-Connection Libraries

    Call-level interface (CLI)

    JDBC

    PHP

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.

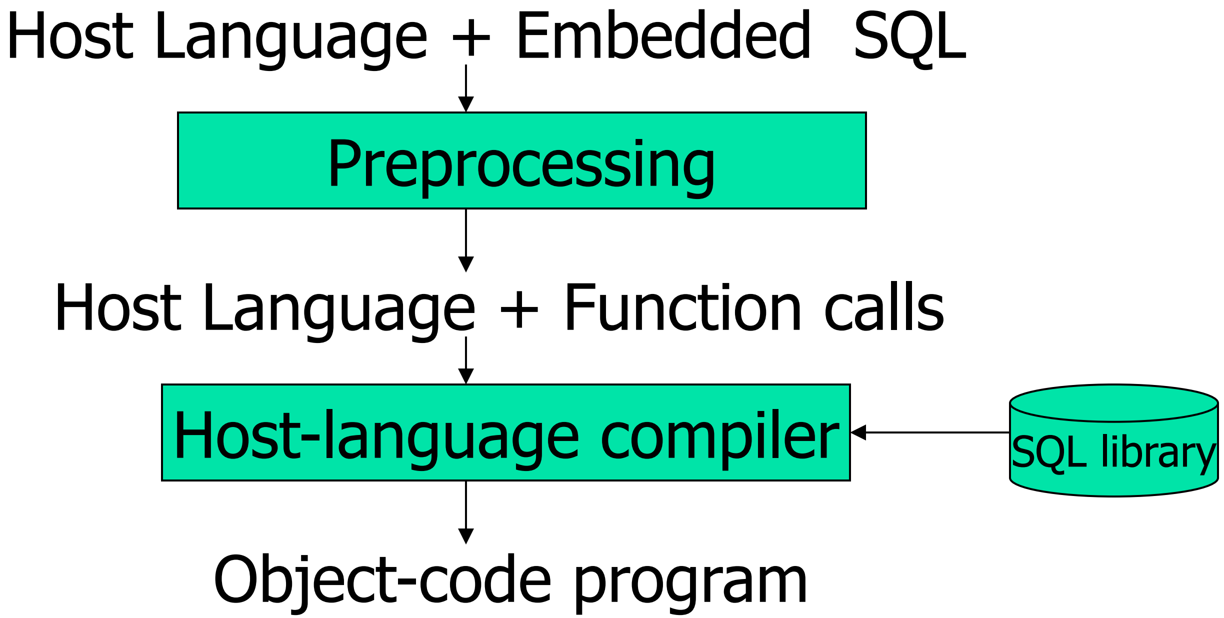
Options

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

SQL in a Programming Environment

- Embedded SQL: add to a conventional programming language (C for example, we called host language), certain statements that represent SQL operation.
- Host language+embedded SQL → code?

System Implementation



- How to identify SQL statements?
- How to move data between SQL and a conventional programming language?
- Mismatch problem exists?

How to recognize SQL statements (the Interface between SQL statements and programming language)

- Each embedded SQL statement introduced with EXEC SQL
- Shared variables : exchange data between SQL and a host language. When they are referred by a SQL statement, these shared variables are prefixed by a colon, but they appear without colon in host-language statements.
- EXEC SQL BEGIN / END DECLARE SECTION to declare shared variables.

the Interface between SQL statements and programming language

- SQL define an array of characters SQLSTATE that is set every time the system is called.
- SQLSTATE connects the host-language program with the SQL execution system.

✓ 00000: no error

✓ 02000: could not be found

Implementations of SQLSTATE

SQL defines an array of characters SQLSTATE that is set every time the system is called.

- Errors are signaled there
- Different systems use different way
- Oracle provides us with a header file sqlca.h that declares a communication area and defines macros to access it, such as NOT FOUND.
- Sybase provides SQLCA with sqlcode

  0:success, <0: fail, 100: not found

Example: Find the price for a given beer at a given bar

Sells (bar, beer, price)

```
EXEC SQL BEGIN DECLARATION SECTION
CHAR theBar[21], theBeer[21];
Float thePrice;
EXEC SQL END DECLARAE SECTION
EXEC SQL SELECT price INTO :thePrice
FROM sells
WHERE beer = :theBeer AND bar =:theBar;
```

Queries produce sets of tuples as a result, while none of the major host languages supports a set data type directly. So, cursors are used.

- A cursor declaration: EXEC SQL DECLARE <cursor> CURSOR FOR <query>
- A statement EXEC SQL OPEN<cursor> : the cursor is ready to retrieve the first tuple of the relation over which the cursor ranges.
- EXEC SQL FETCH FROM < cursor > INTO <list of variables>
- EXEC SQL CLOSE <cursor>: the cursor is no longer ranges over tuples of the relation.

Cursor Example

```
Void worthRanges() {
int i,digits, counts[15];
EXEC SQL BEGIN DECLARE SECTION;
  int worth; char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE execCursor CURSOR FOR
    SELECT netWorth FROM MovieExec;
EXEC SQL OPEN execCursor;
      while (1) { EXEC SQL FETCH FROM execCursor
    INTO :worth;
    if (NO_MORE_TUPLES) BREAK;
    else …..
      }
    EXEC SQL CLOSE execCursor;
…
}
```

More about cursor:

- The order in which tuples are fetched from the relation can be specified.
- The effect of changes to the relation that the cursor ranges over can be limited.
- The motion of the cursor through the list of tuples can be varied.

Modification by cursor

- With Where clause WHERE CURRENT OF followed by the name of the cursor.

Define
NO_MORE_TUPLES !(strc
mp(SQLSTATE,”02000”))

```
L OPEN execCursor;
EXEC SQL FETCH FROM execCursor
INTO :exec ame,:execAddr,:certNo,:worth;
if (NO_MORE_TUPLES) BREAK;
IF (WORTH < 1000)
EXEC SQL DELETE FROM MovieExec
WHERE CURRENT OF execCursor;
else …..
EXEC SQL CLOSE execCursor;
```

Protecting against concurrent updates

```
EXEC SQL DECLARE execCursor INSENSITIVE
CURSOR FOR
SELECT netWorth FROM MovieExec;
```

- The SQL system will guarantee that changes to relation MovieExec made between one opening and closing of execCursor will not affect the set of tuples fetched.
- Insensitive cursors could be expensive, systems spend a lot of time to manage data access.

Scrolling Cursors

- EXEC SQL DECLARE execCursor SCROLL CURSOR FOR MovieExec;

  The cursor may be used in a manner other than moving forward in the order of tuples.

- Follow FETCH by one of several options that tell where to find the desired tuple. Those options are NEXT, PRIOR, FIRST, LAST and so on.

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.
- Sometimes we don’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;
}
```

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 Stored in the Schema

- Aim

  Provide a way for the user to store with a database schema some functions or procedures that can be used in SQL queries or other SQL statements.

Creating PSM Functions and Procedures

Procedure Declarations

```
CREATE PROCEDURE
    <name>(<arglist>)
   local declarations;
   procedure body;
```

Function Declarations

```
CREATE FUNCTION <name> (<parameters>)
  RETURNS <type>
  local declarations
  function body;
```

Example:

```
CREATE PROCEDURE move (
     IN  oldAddr   VARCHAR [255],
     IN  newAddr VARCHAR [255]
     UPDATE MOVIEsTAR
      SET address = newAddr
     WHERE address = oldAddr; )
```

– The parameters of a procedure are triples of mode-name-type

- IN = procedure uses value, does not change value.
- OUT = procedure changes, does not use.
- INOUT = both.

Function Declaration

- Function parameter may only be of mode IN, the only way to obtain information from a function is through its return-value.

Example: Stored Procedure

- Let’s write a procedure that takes two arguments *b* and *p*, and adds a tuple to Sells 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);
```

Where to call?

CALL <procedure name> (<argument list>);

- From a host-language program, e.g.

```
EXEC SQL CALL foo(:x,3);
```

- As a statement of another PSM function or procedure
- As an SQL command issued to the generic SQL interface, e.g. CALL foo(1,3)

Invoking Functions

- It is not permitted to call a function.
- Use the function name and suitable arguments as part of an expression.
- Functions used in SQL expressions where a value of their return type is appropriate.

Simple statements in PSM

- Return statement in a function: RETURN <expression>;
- declare local variables : DECLARE <name><type>;
- Assignments: SET <variable>=<expression>;
  - SET b = 'Bud';
- Groups of statements: BEGIN…END

Separate by semicolons.

- Branching statements: If then else,
- Loops: for-loops, loops,

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

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;
  ```

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;
```

Exceptions i

```
CREATE FUNCTI
   RETURNS INT
DECLARE Not_I
   ‘02000’;
DECLARE Too_Ma
   ‘21000’;
BEGIN
  DECLARE EXIT HANDLER FOR
   Not_Found,Too_Many
    RETURN NULL;// handler declaration
  RETURN (SELECT year FROM Movie WHERE
   title=t);
END;
```

Where to go:

1) continue:execute the statement after the one that raised the exception.
2) Exit:leave the BEGIN…END block.the statement after the block is executed next.
3) Undo: not executed the statement within the block and exit like 2)

Components of Exception handler in PSM

- A list of exception conditions that invoke the handler when raised.
- Code to be executed when one of the associated exceptions is raised.
- An indication of where to go after the handler has finished its work.

DELARE <where to go> HANDLER FOR <condition list> <statement>

Example: Cursor in PSM

- 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 it’s price at Joe’s Bar by $1.