SQL Application Programming

Persistent Stored Modules (PSM)

Peter Scheuermann

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.

Interactive vs. Non-Interactive SQL

- ***Interactive SQL***: SQL statements input from terminal; DBMS outputs to screen
  - Inadequate for most uses
    - It may be necessary to process the data before output
    - Amount of data returned not known in advance
    - SQL has very limited expressive power (not Turing-complete)
- ***Non-interactive SQL***: SQL statements are included in an application program written in a host language, like C, Java, COBOL
  - Nowadays also: as embedded in dynamic webpages

- ***Client-side* vs. *Server-side*** application development
  - Server-side: Stored Procedures and Triggers

SQL in Application Code

- SQL commands can be called from within a *host language* (e.g., C++ or Java) program.
  - SQL statements can refer to host variables (including special variables used to return status).
  - Must include a statement to *connect* to the right database.

- **Two main integration approaches**:
  - **Statement-level interface (SLI)**
    - Embed SQL in the host language (Embedded SQL in C, SQLJ)
    - Application program is a mixture of host language statements and SQL statements and directives
  - **Call-level interface (CLI)**
    - Create special API to call SQL commands (JDBC, ODBC, PHP, …)
    - SQL statements are passed as arguments to host language (library) procedures / APIs

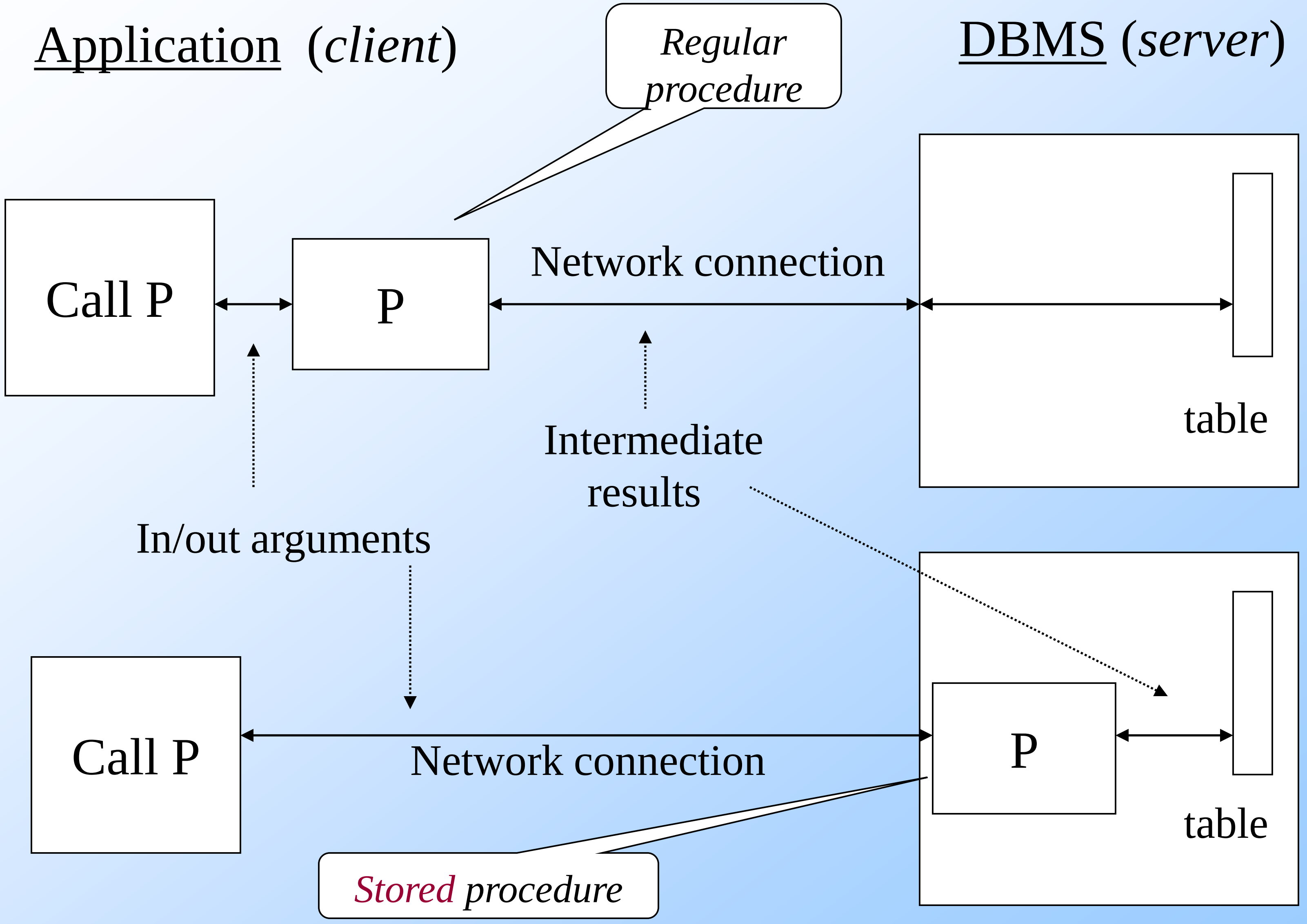
SQL/PSM (persistant stored modules)

- Stored Procedures not only have full access to SQL—store procedures as database schema elements
- All major database systems provide extensions of SQL to a simple, general purpose language
  - SQL:1999 Standard: SQL/PSM
  - Oracle: PL/SQL (syntax differs!!!)
- Extensions
  - Local variables, loops, if-then-else conditions
- Example:

```
CREATE PROCEDURE ShowNumberOfEnrolments
   SELECT uosCode, COUNT(*)
     FROM Enrolled
    GROUP BY uosCode
```

- Calling Stored Procedures: CALL statement
  - Example: `CALL ShowNumberOfEnrolments();`

Stored Procedures



Application (*client*)

DBMS (*server*)

Regular procedure

Call P

P

Network connection

table

Intermediate results

In/out arguments

Call P

Network connection

P

table

Stored procedure

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;  ←  If this statement is executed . . .
      . . .
END LOOP;
 ←  Control winds up here
```

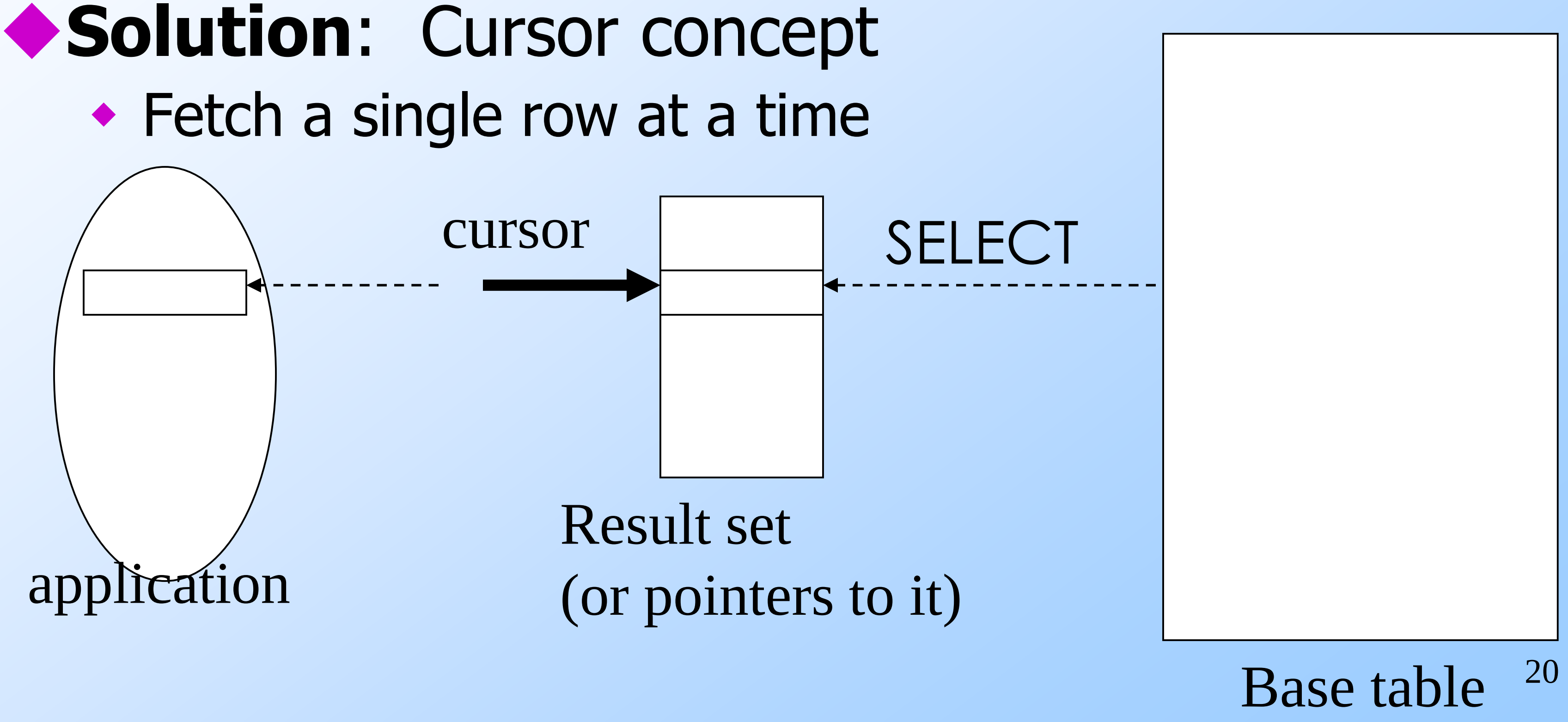
Other Loop Forms

-
  ```
  WHILE <condition>
      DO <statements>
  END WHILE;
  ```

-
  ```
  REPEAT <statements>
      UNTIL <condition>
  END REPEAT;
  ```

Buffer Mismatch Problem
(also: Impedance Mismatch)

- **Problem**: SQL deals with tables (of arbitrary size); host language program deals with fixed size buffers
  - How is the application to allocate storage for the result of a SELECT statement?
- **Solution**: Cursor concept
  - Fetch a single row at a time



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,…,*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)

- Example: We can declare condition NotFound to represent 02000 by:

```
DECLARE NotFound CONDITION FOR
    SQLSTATE ’02000’;
```

- We may declare a *condition*, which is a boolean variable that is true if and only if SQLSTATE has a particular value.

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.