Lecture 17: Systems Aspects of SQL

Wednesday, November 1, 2000

Outline

- Embedded SQL (7.1)
- Transactions in SQL (7.2)

Embedded SQL

- direct SQL (= ad-hoc SQL) is rarely used
- in practice: SQL is embedded in some application code
- SQL code is identified by special syntax

Impedance Mismatch

- Example: SQL in C:
  - C uses int, char[..], pointers, etc
  - SQL uses tables
- Impedance mismatch = incompatible types

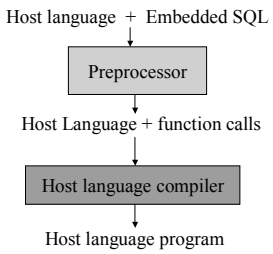
The Impedance Mismatch Problem

Why not use only one language?

- Forgetting SQL: "we can quickly dispense with this idea" [Ullman & Widom, pg. 363].
- SQL cannot do everything that the host language can do.

Solution: use cursors

Programs with Embedded SQL



Interface: SQL / Host Language

Values get passed through shared variables.

Colons precede shared variables when they occur within the SQL statements.

EXEC SQL: precedes every SQL statement in the host language.

The variable SQLSTATE provides error messages and status reports (e.g., "00000" says that the operation completed with no problem).

```
EXEC SQL  BEGIN DECLARE SECTION;
              char  productName[30];
EXEC SQL  END DECLARE SECTION;
```

Example

Product (pname, price, quantity, maker)
Purchase (buyer, seller, store, pname)
Company (cname, city)
Person(name, phone, city)

Using Shared Variables

```
Void  simpleInsert() {

   EXEC SQL BEGIN DECLARE SECTION;
         char    n[20], c[30];          /* product-name, company-name */
         int     p, q;                  /* price, quantity */
         char    SQLSTATE[6];
   EXEC SQL END DECLARE SECTION;

       /*   get values for name, price and company  somehow   */

  EXEC SQL INSERT  INTO Product(pname, price, quantity, maker)
       VALUES  (:n, :p, :q, :c);
}
```

Single-Row Select Statements

```
int  getPrice(char *name) {

  EXEC SQL BEGIN DECLARE SECTION;
           char n[20];
           int p;
           char    SQLSTATE[6];
  EXEC SQL END DECLARE SECTION;

  strcpy(n, name);   /* copy name to local variable */

  EXEC SQL       SELECT price  INTO :p
                 FROM    Product
                 WHERE Product.name = :n;
  return p;
}
```

Cursors

1. Declare the cursor
2. Open the cursor
3. Fetch tuples one by one
4. Close the cursor

Cursors

```
void product2XML() {
 EXEC SQL BEGIN DECLARE SECTION;
          char n[20], c[30];
          int p, q;
          char    SQLSTATE[6];
 EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE crs CURSOR FOR
          SELECT pname, price, quantity, maker
          FROM    Product;

 EXEC SQL OPEN crs;
```

Cursors

```
printf("<allProducts>\n");
while (1) {
    EXEC SQL FETCH FROM crs INTO :n, :p, :q, :c;
    if (NO_MORE_TUPLES) break;
    printf("   <product>\n");
    printf("      <name>     %s   </name>\n",    n);
    printf("      <price>    %d   </price>\n",   p);
    printf("      <quantity> %d   </quantity>\n", q);
    printf("      <maker>    %s   </maker>\n",   c);
    printf("   </product>\n");
}
EXEC SQL CLOSE crs;
printf("</allProducts>\n");
}
```

- What is NO_MORE_TUPLES ?

```
#define NO_MORE_TUPLES !(strcmp(SQLSTATE,"02000"))
```

More on Cursors

- cursors can modify a relation as well as read it.
- We can determine the order in which the cursor will get tuples by the ORDER BY keyword in the SQL query.
- Cursors can be protected against changes to the underlying relations.
- The cursor can be a scrolling one: can go forward, backward +n, -n, Abs(n), Abs(-n).

Dynamic SQL

- So far the SQL statements were visible to the compiler
- In dynamic SQL we have an arbitrary string that represents a SQL command
- Two steps:
  - Prepare: compiles the string
  - Execute: executes the compiled string

Dynamic SQL

```
Void someQuery() {
EXEC SQL BEGIN DECLARE SECTION;
char *command="UPDATE Product SET quantity=quantity+1 WHERE name='gizmo'";
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE myquery FROM :command;

EXEC SQL EXECUTE myquery;
}
```

myquery = a SQL variable, does not need to be prefixed by ":"

Transactions in SQL

serializability

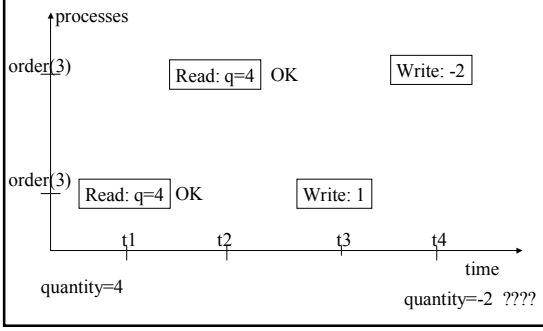
- In real life, many updates happen simultaneously on the database
- Want to ensure that each has an effect equivalent to running alone

Transactions: Serializability

```
int order(char *prod, int quant) {
   EXEC SQL BEGIN DECLARE SECTION;
          char   n[20], c[30];          /* product-name, company-name */
          int    p, q;                  /* price, quantity */
          char   SQLSTATE[6];
   EXEC SQL END DECLARE SECTION;

   EXEC SQL SELECT quantity INTO :q
                FROM   Product WHERE name=:n
   if (quant > q) return (-1);   /* error: not enough items */
   q = quant;
   EXEC SQL UPDATE Product SET quantity =quantity-:q WHERE name=:n;
}
```

Transactions: Serializability



Transactions: Serializability

- a function execution is ***serial*** if it is completely before or completely after any other function's execution
- a function execution is ***serializable*** if it equivalent to one that is serial
- DBMS can offer serializability guarnatees

Transactions: Atomicity

atomicity

- the danger: some database operation may leave the database in an unacceptable state, because of:
  - concurrent execution with other operations
  - Crashes
- Atomicity = an operation either terminates correctly, or does not affect the database

Transactions: Atomicity

```
void order(int q) {
….
EXEC SQL UPDATE Product    SET quantity =quantity-:q
              WHERE …;
```

Crash !

```
EXEC SQL UPDATE Customer SET quantity =quantity+:q
              WHERE …;
```

Transactions: Atomicity

- Can end a database operation in two ways:
  - EXEC SQL COMMIT;
  - EXEC SQL ROLLBACK;