# Lecture 05: SQL Systems Aspects

Friday, October 11, 2002



## Outline

- Embedded SQL (8.1)
- Transactions in SQL (8.6)



## 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

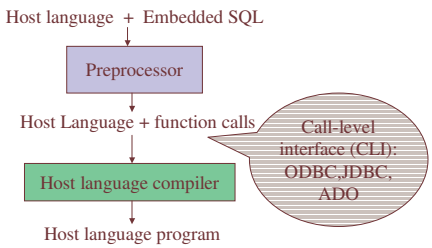
Why not use only one language?

- Forgetting SQL: "we can quickly dispense with this idea" [textbook, pg. 351].
- 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;
```

7

## Example

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

8

## 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);
}
```

9

## 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;
}
```

10

## Cursors

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

11

## 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;
```

12

## 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");
}
EXECT 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

Address two issues:

- Access by multiple users
  - Remember the "client-server" architecture: one server with many clients
- Protection against crashes

## Multiple users: single statements

```
Client 1:
        UPDATE Product
        SET Price = Price – 1.99
        WHERE pname = 'Gizmo'

Client 2:
        UPDATE Product
        SET Price = Price*0.5
        WHERE pname='Gizmo'
```

Two managers attempt to do a discount.
Will it work ?

19

## Multiple users: multiple statements

```
Client 1:  INSERT INTO SmallProduct(name, price)
                SELECT pname, price
                FROM Product
                WHERE price <= 0.99

           DELETE Product
                WHERE price <=0.99

Client 2:  SELECT count(*)
           FROM Product

           SELECT count(*)
           FROM SmallProduct
```

What's wrong ?

20

## Protection against crashes

```
Client 1:
        INSERT INTO SmallProduct(name, price)
                SELECT pname, price
                FROM Product
                WHERE price <= 0.99

        DELETE Product
                WHERE price <=0.99
```

Crash !

What's wrong ?

21

## Transactions

- Transaction = group of statements that must be executed atomically
- Transaction properties: ACID
  - ATOMICITY = all or nothing
  - CONSISTENCY = leave database in consistent state
  - ISOLATION = as if it were the only transaction in the system
  - DURABILITY = store on disk !

22

## Transactions in SQL

- In "ad-hoc" SQL:
  - Default: each statement = one transaction
- In "embedded" SQL:

  BEGIN TRANSACTION
  [SQL statements]
  COMMIT or ROLLBACK (=ABORT)

23

## Transactions: Serializability

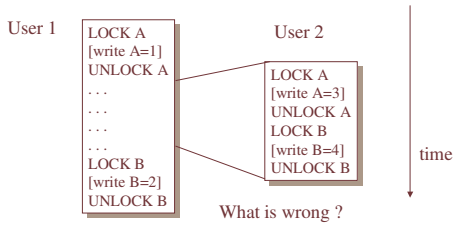
Serializability = the technical term for isolation

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

24

## Serializability

- Enforced with locks, like in Operating Systems !
- But this is not enough:




What is wrong ?

25

## Serializability

- Solution: two-phase locking
  - Lock everything at the beginning
  - Unlock everything at the end
- Read locks: many simultaneous read locks allowed
- Write locks: only one write lock allowed
- Insert locks: one per table

26

## Isolation Levels in SQL

1. "Dirty reads"
   SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
2. "Committed reads"
   SET TRANSACTION ISOLATION LEVEL READ COMMITTED
3. "Repeatable reads"
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
4. Serializable transactions (default):
   SET TRANSACTION ISOLATION LEVEL SERIALIZABLE

Reading assignment: chapter 8.6

27