

SQL in the Real World

- SQL is good for querying, but not so good for complex computational tasks. It is not very good for displaying results nicely.
- Moreover, queries and updates typically occur inside complex computations, for which SQL is not a suitable language.
- Thus, one most often runs SQL queries from host programming languages, and then processes the results.
- One approach: extend SQL.
SQL3 can do many queries that SQL2 couldn't do. But sometimes one still needs to do some operations in a programming language.
- SQL offers two flavors of communicating with a PL:
 - embedded SQL,
 - dynamic SQL.
- Basic rule: if you know SQL, and you know the programming language, then you know embedded/dynamic SQL.

SQL and programming languages cont'd

- Most languages provide an interface for communicating with a DBMS (Ada, C, Java, Cobol, etc).
- These interfaces differ in details, but conceptually they follow the same model.
- We learn this model using C as the host language.
- Examples of difference: SQL statements start and end with
 - EXEC SQL and ; in C
 - EXEC SQL and END-EXEC in Cobol
- SQLSTATE variable: the state of the database after each operation.
It is
 - char, length 6 in C,
 - character, length 5 in Fortran,
 - array [1..5] of char in Pascal

SQL/C interface

- DBMS tells the host language what the state of the database is via a special variable called `SQLSTATE`
- In C, it is commonly declared as `char SQLSTATE[6]`.
- Two most important values:
 - `'00000'` means “no error”
 - `'02000'` means: “requested tuple not found”.The latter is used to break loops.
- Why 6 characters? Because in C we commonly use the function `strcmp` to compare strings, which expects the last symbol to be `'\0'`. Thus we declare `char SQLSTATE[6]` and initially set the 6th character to `'\0'`.

SQL/C interface: declarations

- To declare variables shared between C and SQL, one puts them between `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION`
- Each variable `var` declared in this section will be referred to as `:var` in SQL queries.
- Example:

```
EXEC SQL BEGIN DECLARE SECTION;
    char title[20], theater[20];
    int showtime;
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION
```

Simple insertions

- With these declarations, we can write a program that prompts the user for title, theater, and showtime, and inserts a tuple into Schedule.

```
• void InsertIntoSchedule() {  
  
    /* declarations from the previous slide */  
  
    /* your favorite routine for asking the user for  
       3 values: two strings and one integer.  
       those are put in title, theater, showtime */  
  
    EXEC SQL INSERT INTO Schedule  
           VALUES (:theater, :title, :showtime);  
}
```

- Note how we use variables in the SQL statement:
:theater instead of theater etc.

Simple lookups

- Task: prompt the user for theater and showtime, and return the title (if it can be found), but only if it is a movie directed by Spielberg.

```
• void FindTitle() {  
  
    EXEC SQL BEGIN DECLARE SECTION;  
           char th[20], t1[20];  
           int s;  
           char SQLSTATE[6];  
    EXEC SQL END DECLARE SECTION;  
  
    /* get the values of theater and showtime  
       and put them in variables th and s */
```

Simple lookups cont'd

```
EXEC SQL SELECT title
        INTO   :t1
        FROM   Schedule S, Movies M
        WHERE  S.title = M.title AND
              M.director = 'Spielberg' AND
              S.showtime = :s;
```

```
if (strcmp(SQLSTATE,"02000") != 0)
    printf("title = %s\n", t1)
    else printf("no title found\n");
```

```
}
```

- The comparison `strcmp(SQLSTATE,"02000")` checks if the DBMS responded by saying 'no tuples found'. Otherwise there was a tuple, containing the value of title, and we print it.

Single-value queries

- Those often involve aggregation.
- How many movies were directed by a given director?

```
int count_movies (char director[20]) {
    EXEC SQL BEGIN DECLARE SECTION;
        char dir[20], SQLSTATE[6];
        int m_count;
    EXEC SQL END DECLARE SECTION;

    for (i=0; i<20; ++i) dir[i]=director[i];

    EXEC SQL SELECT COUNT(DISTINCT Title)
            INTO :m_count
            FROM Movies
            WHERE Director = :dir;
    if (strcmp(SQLSTATE, "00000") != 0)
        {printf("error\n"); m_count = 0};

    return m_count;
}
```

Cursors

- Single-tuple insertions or selections are rare when one deals with DBMSs: SQL is designed to operate with tables.
- However, programming languages operate with variables, not tables.
- Mechanism to connect them: **cursors**.
- Cursor allows a program to access a table, one row at a time.
- A cursor can be declared for a table in the database, or the result of a query.
- Variables from a program can be used in queries for which cursors are declared if they are preceded by a colon.

Operators on cursors

- Cursors first must be declared.
- For a table:

```
EXEC SQL DECLARE C_movies CURSOR FOR Movies;
```
- For a query:

```
EXEC SQL DECLARE C_th CURSOR FOR
  SELECT S.theater
  FROM Schedule S, Movies M
  WHERE S.title = M.title
```
- For a query that depends on a parameter:

```
EXEC SQL DECLARE C_th_dir CURSOR FOR
  SELECT S.theater
  FROM Schedule S, Movies M
  WHERE S.title = M.title and M.director = :dir;
```

Operations on cursors

- Open cursor:

```
EXEC SQL OPEN C_movies;
```

```
EXEC SQL OPEN C_th_dir;
```

- The effect of opening a cursor: it points at the first tuple in the table (in this case, either the first tuple of `Movies`), or the first tuple of the result of

```
EXEC SQL DECLARE C_th_dir CURSOR FOR
    SELECT S.theater
    FROM Schedule S, Movies M
    WHERE S.title = M.title and M.director = :dir;
```

- Close cursor:

```
EXEC SQL CLOSE C_movies;
```

Operations on cursors cont'd

- Fetch – retrieves the value of the current tuple and assigns fields to variables from the host language.

- Syntax:

```
EXEC SQL FETCH <cursor> INTO <variables>
```

- Examples:

```
EXEC SQL FETCH C_th_dir INTO :th;
```

fetches the current value of theater to which the cursor `C_th_dir` points, puts the value in `th`, and moves the cursor to the next position.

- If there are multiple fields:

```
EXEC SQL FETCH C_Movies INTO :t1, :dir, :act, :length;
```

Fetches the current (`t1`, `dir`, `act`, `length`) tuple from `Movies`, and moves to the next tuple.

Operations on cursors cont'd

- Other flavors of FETCH:
 - FETCH NEXT: move to the next tuple after fetching the values. This is the default, NEXT can be omitted.
 - FETCH PRIOR: move to the prior tuple after fetching the values.
 - FETCH FIRST or FETCH LAST: get the first, or the last tuple.
 - FETCH RELATIVE <number>: says by how many tuples to move forward (if the number is positive) or backwards (if negative). RELATIVE 1 is the same NEXT, and RELATIVE -1 is the same PRIOR.
 - FETCH ABSOLUTE <number>: says which tuple to fetch. ABSOLUTE 1 is the same FIRST, and ABSOLUTE -1 is the same LAST.

Using cursors: Example

Prompt the user for a director, and show the first five theaters playing the movies of that director.

```
void FindTheaters() {  
  
    int i;  
    EXEC SQL BEGIN DECLARE SECTION;  
        char dir[20], th[20], SQLSTATE[6];  
    EXEC SQL END DECLARE SECTION;  
  
    /* somewhere here we got the value for dir */  
  
    EXEC SQL DECLARE C_th_dir CURSOR FOR  
        SELECT S.theater  
        FROM Schedule S, Movies M  
        WHERE S.title = M.title and M.director = :dir;
```

Using cursors: Example

```
EXEC SQL OPEN C_th_dir;

i=0;
while (i < 5) {
    EXEC SQL FETCH C_th_dir into :th;
    if (NO_MORE_TUPLES) break;
    else printf("theater\t%s\n", th);
    ++i;
}

EXEC SQL CLOSE C_th_dir;
}
```

What is NO_MORE_TUPLES? A common way to define it is:

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

Using cursors for updates

- We consider the following problem. Suppose some lengths of Movies are entered as hours (e.g, 1.5, 2.5), and some as minutes (e.g, 90, 150). We want all to be uniform, say, minutes.
- We assume that no movie is shorter than 5 minutes or longer than 5 hours, so if the length is less than 5, that's an indication that some modification needs to be done.
- Furthermore, we want to delete all movies that run between 4 and 5 hours.

```
void ChangeTime() {
    EXEC SQL BEGIN DECLARE SECTION;
    char t1[20], dir[20], act[20], SQLSTATE[6];
    float length;
    EXEC SQL END DECLARE SECTION;
    EXEC SQL DECLARE C_Movies CURSOR FOR Movies;
```

Using cursors for updates cont'd

```
EXEC SQL OPEN C_movies;
while(1) {
    EXEC SQL FETCH C_Movies INTO
        :tl, :dir, :act, :length;
    if (NO_MORE_TUPLES) break;
    if ( (length > 4 && length < 5) ||
        (length > 240 && length < 300) )
        EXEC SQL DELETE FROM Movies
            WHERE CURRENT OF C_Movies;
    if (length <= 4)
        EXEC SQL UPDATE Movies
            SET Length = 60.0 * Length
            WHERE CURRENT OF C_Movies;
}
EXEC SQL CLOSE C_Movies;
}
```

Other embedded SQL statements

- Connecting to a database:

```
strcpy(db_name, "csc343h");
EXEC SQL CONNECT TO :db_name;
```

- If user names and passwords are required, use:

```
EXEC SQL CONNECT TO :db_name USER :userid USING :passwd;
```

- Disconnecting:

```
EXEC SQL CONNECT RESET;
```

- Save all changes made by the program:

```
EXEC SQL COMMIT;
```

- Rollback (for unsuccessful termination):

```
EXEC SQL ROLLBACK;
```

Dynamic SQL

- Programs can construct and submit SQL queries at runtime
- Often used for updating databases
- General idea:
 - First, an SQL statement is given as a string, with some placeholders for values not yet known.
 - When those values become known, a query is formed, and
 - when it's time, it gets executed.
- Example: a company that fires employees by departments.
First, start getting ready:

```
sqldelete = "delete from Empl where dept = ?";  
EXEC SQL PREPARE dynamic_delete FROM :sqldelete;
```

Dynamic SQL cont'd

- At some later point, the value of the unlucky department becomes known and put in `bad_dept`. Then one can use:

```
EXEC SQL EXECUTE dynamic_delete USING :bad_dept;
```
- May be executed more than once:

```
EXEC SQL EXECUTE dynamic_delete USING :another_bad_dept;
```
- Immediate execution:

```
SQLstring = "delete from Empl where dept='CEO'";  
EXEC SQL EXECUTE IMMEDIATE :SQLstring;
```

Dynamic and Embedded SQL together

- One can declare cursors for prepared queries:

```
my_sql_query = "SELECT COUNT(DISTINCT Title) \
                FROM Movies \
                WHERE Director = ?";
EXEC SQL PREPARE Q1 FROM :my_sql_query;
EXEC SQL DECLARE c1 CURSOR FOR Q1;

/* get value of dir */

EXEC SQL OPEN c1 USING :dir;
EXEC SQL FETCH c1 INTO :count_movies;
EXEC SQL CLOSE c1;
```

- The same operation can be repeated for different values of `dir`.

More than one user

- So far we assumed that there is only one user. In reality this is not true.
- While a cursor is open on a table, some other user could modify that table. This could lead to problems.
- One way of addressing this: *insensitive cursors*.

```
EXEC SQL DECLARE C1
    INSENSITIVE CURSOR FOR
        SELECT Title, Director
        FROM Movies
```

- This guarantees that if someone modifies `Movies` while `C1` is open, it won't affect the set of fetched tuples.
- This is a very expensive solution and is not used very often.

Problems with more than one user

- We have a bank database, with a table Account, one attribute being balance.
- The following is a function that transfers money from one account to another. It must enforce the rule that one cannot transfer more money than the balance on the account.

```
EXEC SQL BEGIN DECLARE SECTION;
    int acct_from, acct_to, balance1, amount;
    char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;

void Transfer() {
    /* we ask the user to enter
       acct_from, acct_to, amount */
```

Problems with more than one user

```
EXEC SQL SELECT balance INTO :balance1
    FROM Account
    WHERE account_id = :acct_from;

if (balance1 < amount)
    printf("Insufficient amount in account %d\n", acct_from)
else {
    EXEC SQL UPDATE Account
        SET balance = balance + :amount
        WHERE account_id = :acct_to;
    EXEC SQL UPDATE Account
        SET balance = balance - :amount
        WHERE account_id = :acct_from;
    }
}
```

Problems with more than one user

- Assume that `acct_to` and `acct_from` are joint accounts, with two people being authorized to do transfers.
- Let `acct_from` have \$1000. Suppose both users try to transfer \$1000 from this account.
- Sequence of events:
 - User 1 initiates a transfer. Condition is checked and the first UPDATE statement is executed.
 - User 2 initiates a transfer. Condition is checked, and met, since the second UPDATE statement from the first transfer hasn't been executed yet. Now both UPDATE statements are executed.
 - User 1's transfer operation is finished.
- `acct_from` has balance $-\$1000$, despite an apparent safeguard against a situation like this.

Transactions and atomicity

- Why did this happen?
- Because the operation wasn't executed atomically.
- Transaction: a group of statements that are executed atomically on a database.
- Declaration sections, and connecting to a database, do *not* start a transaction.
- A transaction starts when the first statement accessing a row in a database is executed (e.g., OPEN CURSOR).
- Normally, a transaction ends with the last statement of the program, but one can end it explicitly by either
EXEC SQL COMMIT; or
EXEC SQL ROLLBACK;

Transactions and atomicity cont'd

- We revisit the `Transfer()` function.
- Transaction starts with

```
EXEC SQL SELECT balance INTO :balance1
        FROM Account
        WHERE account_id = :acct_from;
```

- To ensure that the problems with concurrent execution do not occur, one can state

```
EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

- The meaning of this will become clear when we study transaction processing.
- Fortunately, this is the default, and the statement is not necessary.

Transactions and atomicity cont'd

- If the test (`balance1 < amount`) is true, we may prefer to abort the transaction:

```
if (balance1 < amount) {
    printf("Insufficient amount in account %d\n", acct_from);
    EXEC SQL ROLLBACK;
}
```

- If there is sufficient amount of funds, we can put after `UPDATE` statements

```
EXEC SQL COMMIT;
```

to indicate successful completion.

Transactions and isolation

- Isolation means that to the user it must appear as if no other transactions were running. The basic level of isolation is

```
EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE
```

- There are others, that have to do with dirty reads.
- Dirty read: read data modified by another transaction, but not yet committed.
- One can explicitly specify

```
SET TRANSACTION [READ ONLY | READ WRITE]  
ISOLATION LEVEL READ [COMMITTED | UNCOMMITTED]
```

- READ ONLY or READ WRITE specify whether transaction can write data (READ WRITE is the default and can be omitted).
- COMMITTED indicates that dirty reads are not allowed (only committed data can be read).

Dealing with errors

- SQL92 standard specifies a structure SQLCA (SQL Communication Area) that needs to be declared by
EXEC SQL INCLUDE SQLCA;
- The main parameter is `sqlca.sqlcode`, where 0 indicates success.
- SQL99 eliminates SQLCA, and many programs have the following structure:

```
...  
EXEC SQL CONNECT TO...  
EXEC SQL WHENEVER SQLERROR goto do_rollback;  
while (1) {  
    /* loop over tuples */  
    /* operations that assume successful execution */  
    continue;  
do_rollback:  
    EXEC SQL ROLLBACK; /* other operations, e.g. printing */  
}  
EXEC SQL DISCONNECT CURRENT;
```