Lecture Notes

Database Management

Patrick E. O'Neil and Elizabeth O'Neil

Chapter 5: Embedded SQL Programs.

Embedded SQL means SQL statements embedded in *host language* (C in our case). The original idea was for end-users to access a database through SQL. Called *casual users*.

But this is not a good idea. Takes too much concentration. Can you picture airline reservation clerk doing job with SQL? Customers waiting. Booking flight (update of seat table). Billing?
 - Need to know all tables & columns, create complex syntax (FOR ALL)
 - too much risk of mistakes, especially with updates

Instead, we have an *Application Programmers* create menu applications, perform selects and updates programmatically.

Programmers can spend a lot of time making sure the right SQL statement is used; programmers are temperamentally suited to this kind of work. Of course nice to have interactive SQL for some situations.

Aim for this chapter is to be able to implement ANY CONCEIVABLE ALGORITHM using a C program that can get at data through SQL statements.

Gets pretty complicated, but don't get lost -- there's a good reason for each new piece of complexity if you just understand it.

5.1 SQL statements in C (general host language) have slightly different syntax than the SQL we've seen so far.

```
exec sql select count(*) into :host_var from customers;
```

Statement starts with exec sql. Variable (host_var) needed to receive answer, so new Into clause; colon shows DBMS this is a program variable.

C compiler doesn't recognize SQL statements. The "exec sql" phrase is a signal to an explicit SQL precompiler that turns SQL into real C calls.

(Already have precompiler acting first when give gcc command for things like #include, #define; this is a different one).

In Oracle, start with pgm.pc (meaning has embedded SQL statements: exec sql . . .). Then give command (See Appendix B)

```
proc iname=pgm.pc
```

Creates new source file, pgm.c, with exec sql statements turned into pure C: they are now calls to ORACLE monitor functions.

Next do true compilation and linkage (with prompt.c, for example), create executable program pgm.ox. Have makefile. WIll create pgm.o:

```
gcc -g -c try.c
```

And so on. The provided makefile knows all the details. You can do everything in the homework using the makefile included. Just create your pgm.pc and type:

```
make E=pgm    (NO spaces on either side of "="; pgm.pc must exist)
```

5.1.1. A simple program.

Declare section. Need to declare C variables so they are understandable to ORACLE (ORACLE needs to know type of variable). Here is how we would set a variable used in a search condition and select into two others.

```
cust_id = "c001"; /* or prompt user for cust id */
exec sql select cname, discnt into :cust_name, :cust_discnt
    from customers where cid = :cust_id;
```

Note use of colon when variable appears in various positions in select statement. Important in Where clause so can tell :cust_id isn't a constant.

At runtime, cust_name and cust_city will be filled in with values "TipTop" and "Duluth". Could now write in program:

```
printf("Customer name is %s, and customer city is %s\n",
    cust_name, cust_city);
```

Note don't use colons here (used in exec sql as hint to dumb precompiler). In order to use C variables in SQL, must put in Declare Section:

```
    exec sql begin declare section;
        char  cust_id[5]  =  "c001",  cust_name[14];
        float  cust_discnt;
    exec sql end declare section;
```

The character arrays need to contain enough characters to hold string values of columns PLUS ONE for terminal '\0' character. E.g., cid was declared as column char(4), with values like 'c001' in SQL.

In C when we specify a string such as "c001" (note double quotes), this includes a terminal zero value ('\0') at the end: this serves as a signal to functions such a printf that the string has ended.

Conversion can also occur if declare

```
        int  cust_discnt;
```

in Declare section. Then float value 10.00 will become 10, but 10.50 will also become 10 — lose fractional significance.

SQL connect and disconnect. Can use sqlplus Scott to attach,

```
  strcpy(username, "Scott");           /* set up username and password      */
  strcpy(password, "Tiger");           /* for Oracle login                  */
  exec sql connect :username identified by :password;
```

This is what is used in ORACLE (Note: can't use literal names, must be in character string. Note too that COUNTED string (in text) is NOT needed!!!!)

The command to disconnect is represented by:

```
    exec sql commit release;   /* ORACLE disconnect */
```

<u>Now look at Example 5.1.1, Figure 5.1, pg 207</u>. Task is to prompt repeatedly for customer cid and print out the customer name and city. Halt when user inputs a null line (just hits CR).

```
#include  <stdio.h>
```

exec sql include sqlca;

exec sql whenever sqlerror stop;   --covered later

Connect, Loop, prompt, select, commit, print. Commit releases locks.   Loop is while prompt( ) > 0, which is length of token in string input by user (white space doesn't count). When user types CR, end loop.

How is ORACLE program 5.1 different from SQL-92 Standard?

In ORACLE can't use literal name in "exec sql connect". must be in a character string. Also must include password.

See prompt() function explained on page 269 and also in Appendix B. Expected to use in homework (available online, handled by makefile).

Put out prompt[ ] string to terminal to insert any number K of tokens (separated one from another by whitespace: '\n', ' ' (SP), '\t'). Format:

```
    int prompt(char prompt_str[],int ntokens, ...);
```

Variable-length argument list represented by ellipsis (`...`) (See K&R). Argument ntokens shows number of tokens to input.

Ellipsis (. . .) represents a variable number of argument pairs: buf1, len1, buf2, len2, . . ., bufN, lenN.

Here, bufK, K = 1 to N, is a character array to contain the Kth token (always char str), and lenK is the maximum allowed length of the Kth token.

All tokens will be read into character arrays by the prompt()

If some of these should be interpreted as int or float numbers, use ANSI stanard C library function sscanf(), with appropriate conversion string:

E.g.: %d for decimal integer and %f for double or float.

Can look at code: uses fgets, sscanf. Could use scanf but multiple tokens on a line (input by confused user) will cause things to get out of synch.

fgets brings keyboard input into array line; sscanf parses K tokens into x. (Any other tokens are lost.) On pg, 216 is prompt2 if need to input two values on one line.

OK, up to now, retrieved only a single row into a variable:

```
     exec sql select cname, discnt into :cust_name, :cust_discnt
          from customers where cid = :cust_id;
```

Selecting multiple rows with a Cursor. (pg. 270, program in **Fig 5.2, pg. 273**)

The Select statement used above can only retrieve a SINGLE value. To perform a select of multiple rows, need to create a CURSOR.

It is an Error to use array to retrieve all rows.

ONE-ROW-AT-A-TIME PRINCIPAL: Need to retrieve one row at a time: cursor keeps track of where we are in the selected rows to retrieve.

COMMON BEGINNER'S MISTAKE, to try to get all the data FIXED in an array. Find the max of an input sequence of integers (don't know how many).

Here: Declare a cursor, open the cursor, fetch rows from the cursor, close the cursor. E.g., declare a cursor to retrieve aids of agents who place an order for a given cid, and the total dollars of orders for each such agent.

```
     exec sql declare agent_dollars cursor for
          select aid, sum(dollars) from orders
          where cid = :cust_id group by aid;
```

But this is only a declaration. First Runtime statement is Open Cursor:

```
     exec sql open agent_dollars;
```

VERY IMPORTANT: when open cursor, variables used are evaluated at that moment and rows to be retrieved are determined.

If change variables right after Open Cursor statement is executed, this will not affect rows active in cursor.

Now fetch rows one after another:

```
exec sql fetch agent_dollars into :agent_id, :dollar_sum;
```

The Into clause now associated with fetch rather than open, to give maximum flexibility. (Might want to fetch into different variables under certain conditions.)

A cursor can be thought of as always pointing to a CURRENT ROW. When just opened, points to position just before first row. When fetch, increment row and fetch new row values.

If no new row, get runtime warning and no returned value (as with getchar( ) in C, value of EOF).

Note "point before row to be retrieved" better behavior than "read values on row, then increment".

This way, if runtime warning (advanced past end of rows in cursor) then there are no new values to process, can leave loop immediately.

How do we notice runtime warning? SQL Communication Area, SQLCA (now deprecated by SQL-92, but stick with). Recall in pgm. 5.1.1 had:

```
exec sql include sqlca;
```

This creates a memory struct in the C program that is filled in with new values as a result of every exec sql runtime call; tells status of call.

Class 25.

See **Fig 5.2, pg. 273**.

Review: Declare a cursor, open the cursor, fetch rows from the cursor, close the cursor.

E.g., declare a cursor to retrieve aids of agents who place an order for a given cid, and the total dollars of orders for each such agent.

```
exec sql declare agent_dollars cursor for
     select aid, sum(dollars) from orders
     where cid = :cust_id group by aid;
```

But this is only a declaration. First Runtime statement is Open Cursor:

```
exec sql open agent_dollars;
```

When open cursor, variables used are evaluated at that moment and rows to be retrieved are determined.

Now fetch rows one after another:

```
exec sql fetch agent_dollars into :agent_id, :dollar_sum;
```

A cursor can be thought of as always pointing the CURRENT ROW (if any), the row last fetched. When fetch and get NO DATA RETURNED, means end loop right away.

NOTE, in general, we cannot pass a cursor as an argument. But can make it external to any function in a file, so all functions can get at it.

How do we tell that the most recent fetch did not retrieve any data? Remember exec sql include sqlca? Communication area

Decprecated, but sqlca.sqlcode is still an important variable.

Still works to test the value of sqlca.sqlcode after delete, update, insert, etc. to figure out what errors occurred. **MAY** be replaced by SQLSTATE.

ORACLE has implemented SQLSTATE, but still uses sqlca.sqlcode because gives more error codes. Can use SQLSTATE in ANSI version of Oracle.

For portability best to use different method for most things: WHENEVER statement, below.

NOTE that sqlca must be declared where all functions that need to can access it: usually external to any function. Go through logic.

Starting to talk about 5.2 Error handling.

EXEC SQL INCLUDE SQLCA creates a C struct that's rather complicated, and denegrated, but we deal mainly with a single component, sqlca.sqlcode.

This tells whether

```
   sqlca.sqlcode == 0, successful call
         < 0, error, e.g., from connect, database does not exist -16
              (listed as if positive)
         > 0, warning, e.g., no rows retrieved from fetch
              (saw this already, tell when cursor exhausted)
```

Error is often a conceptual error in the program code, so important to print out error message in programs you're debugging. Come to this.

See Example 5.2.4 in text.

In general, there are a number of conditions that a Whenever statement can test for and actions it can take. General form of Whenever statement:

```
      exec sql whenever <condition> <action>
```

Conditions.

o SQLERROR	Tests if sqlca.sqlcode < 0
o NOT FOUND	Tests if no data affected by Fetch, Select, Update, etc.
o SQLWARNING	Tests if sqlca.sqlcode > 0 (different than not found)

Actions

o CONTINUE	Do nothing, default action
o GOTO label	Go to labeled statement in program
o STOP	In ORACLE, Prints out error msg and aborts program
o DO func_call	(IN ORACLE ONLY)Call named function; very useful

Look at exercise due: 5.3. Know how to test for cust_id value that doesn't exist.

The reason the call action is so handy is that WHENEVER statements don't respect function divisions. If we write a single whenever at the beginning of a program with a lot of functions:

```
    whenever sqlerror goto handle_error

    main( ){
    . . . }
    func1( )      {
    . . .}
    func2( )      {
    . . .
```

Using the above trick, the effect of the WHENEVER statement will span the scope of the different functions. However, DO is not portable, and the portable way requires a GOTO target in each function. This GOTO-target code of course can have a function call.

The WHENEVER statement is handled by the precompiler, which puts in tests after all runtime calls (exec sql select... or whatever) and doesn't care about what function it's in (it doesn't even KNOW). For example:

```
    whenever sqlerror goto handle_error;  /* below this, will do this GOTO on error  */
```

Continues doing this until its actions are overridden by a WHENEVER with a different action for the same condition:

```
    whenever sqlerror continue;    /* below this, will do nothing */
```

But note in example above, there must be a label handle_error in all these functions. A call to a handle_error function would save a lot of code.

Question. What happens here, where only two whenever statements are listed explicitly?

```
    main( )
    {
        exec sql whenever sqlerror stop;
        . . .
```

```
         goto label1;
         . . .
         exec sql whenever sqlerror continue;
label1:  exec sql update agents set percent = percent+1;
```

If we arrive at label1 by means of the goto statement, which whenever statement will be in force for condition sqlerror? E.g., what if mispell columname percent? (Answer is: continue.)

What if did:

```
     main( )
     {
         exec sql whenever sqlerror goto handle_error;
         exec sql create table customers
            (cid char[4] not null, cname varchar(13), . . . )
         . . .
handle_error:
         exec sql drop customers;
         exec sql disconnect;
         exit(1);
```

What's the problem? Possible infinite loop because goto action still in force when drop table (may be error). Need to put in:

```
         exec sql whenever sqlerror continue;
```

Right at beginning of handle_error. Overrides goto action (won't do anything). (Default effect if don't say anything about whenever.)

Also need to take default non-action to perform specific error checking. Maybe have alternative action in case insufficient disk space when try to create table:

```
     exec sql whenever sqlerror goto handle_error;
     exec sql create table customers ( . . .)
     if (sqlca.sqlcode == -8112)          /* if insufficient disk space */
         <handle this problem>
```

But this won't work, because esqlc places test

```
     if(sqlca.sqlcode < 0)
         <call stopfn>
```

IMMEDIATELY after create table, so it gets there first and test for sqlca.sqlcode = -8112 never gets entered (< 0 gets there first).

So have to do whenever sqlerror continue before testing for specific error.

Book tells you how to get a specific error msg. Good use in case don't want to just print out error msg. (Discuss application environment for naive user: never leave application environment, write errors to log file).

Recall that if discnt in customers has a null value for a row, do not want to retrieve this row with:

```
    select * from customers where discnt <= 10 or discnt >= 10;
```

But now say we performed this test in program logic:

```
    exec sql select discnt, <other cols> into :cdiscnt, <other vars>
        where cid = :custid;
```

Now we decide to print out these col values in situation:

```
    if(cdiscnt <= 10 || cdiscnt > 10) printf (. . .).
```

Now look! Host variable cdiscnt is a float var, AND ALL MEANINGFUL BIT COMBINATIONS ARE TAKEN UP WITH REAL VALUES. There's no way that cdisnt will fail the if test above.

Need to add some way to test if the variable cdisnt has been filled in with a null value (then include AND NOT NULL in if test). Can do this with indicator variable, retrieve along with discnt:

```
    short int cdiscnt_ind;     /* form of declaration */
```

Then change select:

```
    exec sql select discnt, <other cols> into :cdiscnt:cdiscnt_ind,   /*
ORACLE syntax */
        <other vars> where cid = :custid;
```

Now if cdiscnt_ind == -1, value of cdiscnt is really null. Add test that cdiscnt is not null by:

```
    if((cdiscnt <= 10 || cdiscnt > 10) && cdiscnt_ind <> -1)
        printf (. . .).
```

Note can also STORE a null value by setting a null indicator cdiscnt_ind and writing

```
    exec sql update customers set discnt = :cdiscnt:cdiscnt_ind where . . .
```

One other common use for indicator variable for char column being read in to array, is to notifiy truncation by value > 0, usually length of column string.

Class 26.
Indicator Variables

Section 5.3. Complete descriptions of SQL Statements.

Select. See Section 5.3, pg 281, Figure 5.3. CAN ONLY RETRIEVE ZERO OR ONE ROW.

Look at general form: No UNION or ORDER BY or GROUP BY in Basic SQL form . like Subquery form on pg. 144, add into clause. (How use GROUP BY with only one row retrieved? Still part of full standard)

See Figure 5.4, corresponding (analogouus) C types for column type. Conversion done if integer col type, float C var.

Note that in general a variable can be used to build up an expression in a search_condition:

```
select cname into :cust_name where cid = :custid and city = :cname;
```

But we **cannot use character string to contain part of statement requiring parsing**:

```
char cond[ ] = "where cid = 'c003' and city = 'Detroit'";
exec sql select cname into :custname where :cond;
```

NOT LEGAL. The ability to do this is called "Dynamic SQL", covered later.

Declare Cursor statement, pg. 283. Used when retrieving more than one row in select, so basically an extension of interactive Select.

Has GROUP BY, HAVING, UNION, ORDER BY. Adds clause: for update of, need later.

Cursor can only move FORWARD through a set of rows. Can close and reopen cursor to go through a second time.

Two forms of delete, pg. 283, Searched Delete and Positioned Delete:

```
exec sql delete from tablename [corr_name]
    [where search_condition | where current of cursor_name];
```

After Searched Delete, used to be would examine sqlca.sqlerrd[2] to determine number of rows affected. New way exists (Don't know yet).

After Positioned delete, cursor will point to empty slot in cursor row sequence, like right after open or when have run through cursor. Ready to advance to next row on fetch. Works just right to delete everything after look at it in loop:

```
  LOOP
     exec sql fetch delcust into :cust_id;
     <work>
     exec sql delete from customers where current of delcust;
  END LOOP
```

If cursor moved forward to next row (say) after delete, would only be deleting every OTHER row.

Could create cursor to select all customers in Detroit to delete them, or could select all customers and then check if city is Detroit. First is more efficient. PRINCIPAL: do all search_conditions before retrieving.

But probably a Searched Delete is most efficient of all for Detroit customers -- save HAVING to switch threads, do all in query optimizer.

Second paragraph on p 284: In order for positioned delete to work, the cursor must (1) be already opened and pointing to a real row, (2) be an updatable curson (not READ ONLY), (3) FROM clauses of delete must refer to same table as FROM clause of cursor select.

Two forms of Update statement, pg. 285: Searched Update and Positioned Update. Same ideas as with Delete (sqlca.sqlerrd[ ] will contain count of rows affected).

First printing: the most general Update syntax is on pg. 773, Figures C.33 and C.34.

Same Update form as earlier, pg. 149: each column value can be determined by a scalar expression, an explicit null, or a value given by a scalar subquery.

Insert, pg. 286. No positioned insert, because position determined in other ways.

Open, Fetch, Close, pg. 286. Open evaluates expressions, sets up row list to retrieve from, unchanging even if expression values change.

Create table, drop table, connect, disconnect. Not create database, because that is not an SQL command. SEE APPENDIX C for more info.

Section 5.4 Idea of concurrency, bank tellers have simultaneous access to data. Problem: Example 5.4.1. Inconsistent view of data. Table A of accounts, A.balance where A.aid = A2 will be called A2. balance. One process wants to move money from one account to another. Say $400.00.

S1: A1.balance == $900.00 A2.balance == $100.00

S2: A1.balance == $500.00 A2.balance == $100.00

S3: A1.balance == $500.00 A2.balance == $500.00

If another process wants to do a credit check, adds A1.balance and A2.balance, shouldn't see S2 or may fail credit check. See Example 5.4.1, pg. 245.

Create an idea called transactions. Programming with transactions makes guarantees to programmer, one of which is Isolation. Means every transaction acts as if all data accesses it makes come in serial order with no intervening accesses from others. E.g.:

```
T1:  R1(A1) W1(A1) R1(A2) W1(A2)   (explain notation)
T2:  R2(A1) R2(A2)
```

Turn Update in to R then W. Turn Select into R only. Problem of S2 comes because of schedule:

```
R1(A1) W1(A1) R2(A1) R2(A2) R1(A2) W1(A2)
```

But this is not allowed to happen with transactions, must act like:

T1 then T2 (T2 sees state S3) or T2 then T1 (T2 sees S1).

Don't care which. Called Serializability.

Class 27.

Review idea last time of concurrency.

R1(A1) W1(A1) R2(A1) R2(A2) R1(A2) W1(A2)

But this is not allowed to happen with transactions, must act like:

T1 then T2 (T2 sees state S3) or T2 then T1 (T2 sees S1).

Don't care which. Called Serializability.

(Why do we want to have real concurrency, class? Why not true serial execution? Idea of keeping CPU busy. More next term.)

Two new SQL statements to cause transactions to occur.

exec sql commit work; Successful commit, rows updated, become concurrently visible.

exec sql rollback; Unsuccessful abort, row value updates rolled back and become concurrently visible.

Transactions START when first access is made to table (select, update, etc.) after connect or prior commit or abort. Ends with next commit work or rollback statement or system abort for other reason.

Recall idea that we hold locks, user has SQL statements to say: logical task is complete: you can make updates permanent and drop locks now.

Typically, applications loop around in logic, with user interaction, transaction extends from one interaction to the next. But might have multiple interactions between interactions. DON't typically hold transaction during user interaction (livelock).

User interaction may set off a lot of account balancing: each Tx subtracts money from n-1 acct balances, adds money to an end one. Say should abort if one acct is under balance.

Clearly system can't guess when one set of accts has been balanced off and start immediately on another set. User must tell WHEN to release locks.

Value of rollback is so don't have to write code to reverse all prior changes.

- **Atomicity.** The set of updates performed by a Tx are atomic, that is, indivisible. If something bad happens (terminal goes down) will abort.
- **Consistency.** E.g., money is neither created nor destroyed by logic, then won't happen.
- **Isolation.** As if serial set of Txs.
- **Durability.** Even if lose power & memory (crash), lose place in what doing, will RECOVER from the crash and guarantee atomicity.

LOCKING. Simple version here. (a) When a Tx accesses a row R, first must get lock. (b) Locks are held until Tx ends (Commit). (3 & 4) Locks are exclusive, so second locker will wait if it can.

Example 5.4.2. Recall problem of inconsistent view.

R1(A1) W1(A1) R2(A1) R2(A2) C2 R1(A2) W1(A2) C1

where add Ci for commit. Now add locks.

L1(A1) R1(A1) W1(A1) L2(A1) (conflict, must wait) L1(A2) R1(A2) W1(A2) C1 (Releases T1 locks including L1(A); Now lock request L2(A1) can succeed) R2(A1) R2(A2) C2

Result is T2 sees State S3. No matter what arrangement of attempts, T2 will see either S1 or S3. But New idea: DEADLOCK. What if rearrange:

R1(A1) W1(A1) R2(A2) R2(A1) C2 R1(A2) W1(A2) C1

Now locking proceeds as follows:

L1(A1) R1(A1) W1(A1) L2(A2) R2(A2) L2(A1) (prior L1(A1), so T2 must wait) L1(A2) (prior L2(A2) so T1 must wait: DEADLOCK! illustrate)

(must choose one Tx to abort, say T2, L1(A2) is successful) R1(A2) W1(A2) C2 (Now T2 audit attempt retrys as T3) L3(A2) R3(A2) L3(A1) R3(A1) C3

And state seen by T2 (retried as T3) is S3. If aborted other one, state seen by T2 would be S1. Note that in general, locks held are much more sophisticated than these exclusive locks.

This means that programmer must watch out for deadlock error return, attempt a retry. See Example 5.4.4, pg 296.

Note can't run into a deadlock abort until already HOLD a lock and try to get another one. Of course this could happen in a single instruction in (e.g.) Searched Update with multiple rows. But if only single rows accessed, no

No user interaction during transaction or Livelock. For example, if retrieve data of quantity available for an order and then confer with customer. See Example 5.4.5, program pg. 300But problem as a result.

To avoid this, typically commit transaction after retrieving data, then confer with customer, the start new transaction for update based on customer wishes. But new problem. See Example 5.4.6, program pg 301