EMBEDDED SQL

Introduction

The second half of the English book 'Databases and SQL' describes the use of the programming language SQL for retrieving and updating data in a relational database.
SQL is used in the book as a language that is self-contained. Statements written in SQL do not have to be self-contained. They can also be used as part of programs written in for instance COBOL or C. This is called embedding. We refer to SQL in that form as 'embedded SQL'. In the case the use of SQL in these programming languages, the programming languages like COBOL, C or FORTRAN are host languages.
In this article we describe how SQL is used as an embedded language.

The following will be discussed in this article:
- how we can insert SQL statements between the programming statements of a host program
- how we deal with variables of the host language in the SQL statements
- how we can use the results of queries and update statements
- how we can use transactions in embedded SQL
- how we can use error messages
- overview of the examples of embedding in COBOL

For the examples we will use the example databases you can find in the book 'Databases and SQL' (English edition), paragraph 6.2:
- labour division database
- library database

Embedding and host variables

Separating SQL statements from the host program
It is important that the compiler of the host program recognizes the SQL statements and passes these on to SQL itself. A COBOL compiler expects that every SQL statement is preceded by:

```
EXEC SQL
```

and is ended with:

```
END-EXEC.
```

example

```
EXEC SQL
UPDATE LOCATION
SET CITY = 'WICKFORD'
WHERE LOCNAME = 'FASTBITE'
END-EXEC.
```

host variables in SQL statements
In the SQL statements we can use variables of the host program. These host variables are preceded by a colon in order to distinguish them from column names of tables:

```
:hostvariablename
```

example

```
EXEC SQL
DELETE FROM LOCATION
WHERE LOCNAME = :LOCNAME
END-EXEC.
```

Queries and update statements

using a cursor
A host program can only work with one row from a table at a time. A SELECT statement however can produce more than one row. The host program therefore needs a mechanism to use these rows (from a table the SELECT statement produces) separately. This mechanism is a kind of pointer that points to one specific row of the table the SELECT statement has produced. Such a pointer is called a cursor. This cursor enables the host program to use only the values of the row the cursor points to.

The output of a SELECT statement can only be used with a cursor (an exception to this that we will discuss further on is: SELECT INTO). A cursor has to be declared, opened, used and closed with separate statements.

The statement for declaring the cursor is:

```
DECLARE cursorname CURSOR FOR select-statement
```

(lower case letters signify where something has to be filled in)

example

```
EXEC SQL
DECLARE CUR1 CURSOR FOR
      SELECT E1.ENR, E1.SALARY,
             E2.ENR, E2.SALARY
      FROM EMPLOYEE E1,
             EMPLOYEE E2,
             REPLACEMENT R
      WHERE E1.ENR  = R.REPLACED
      AND R.SUBSTITUTE = E2.ENR
      AND E1.FNAME = 'DIRECTOR'
      ORDER BY E1.FNAME
END-EXEC.
```

© copyright Ton de Rooij

retrieved from www.tonderooij.com

opening a cursor
The statement for opening/initializing the cursor is:

```
OPEN cursorname
```

example

```
EXEC SQL
OPEN CUR1
END-EXEC.
```

statement FETCH
The statement for retrieving rows with a cursor is:

```
FETCH cursorname INTO :variable1
                  [,variable2]...
```

In the FETCH statement there must be exactly as many variables as that the SELECT statement delivers columns in the output. In addition to that the definition of the variables must be done in a way that they can contain the offered values. For instance the variable must be defined as a text column in case text is read.
Apart from reading a row that is addressed by a cursor, the row can also be updated and even be deleted.

example

```
EXEC SQL
FETCH CUR1
      INTO  :REPLACED,
            :SALARY1,
            :SUBSTITUTE,
            :SALARY2
END-EXEC.
```

statement UPDATE
For updating the following statement is used:

```
UPDATE tablename
SET setdescription
WHERE CURRENT OF cursorname
```

example
(the example uses a cursor CUR3 that reads one by one the rows of the table EMPLOYEE. The employee examined gets after the host program decides that the employee qualifies a pay rise)

```
EXEC SQL
UPDATE EMPLOYEE
SET SALARY =
SALARY * (1+
  ( :RISEPERCENTATGE / 100))
WHERE CURRENT OF CUR3
END-EXEC.
```

statement DELETE
The statement for deleting the row that the cursor points to, is:

```
DELETE FROM tablename
WHERE CURRENT OF cursorname
```

(the table names that are mentioned in the UPDATE and DELETE statements have to align with the table the cursor points at. This is only possible in case the cursor of a SELECT statement has a FROM clause with only one table. In addition to that the output of the SELECT statement must refer to only one row of the table)

example
(in the example the cursor refers one by one to the rows of the table EMPLOYEE. The host program decides which employees must be deleted. If this is the case for a certain employee the following statement is executed)

```
EXEC SQL
DELETE FROM EMPLOYEE
WHERE CURRENT OF CUR6
END-EXEC.
```

ending the use of the cursor
The statement to end the execution of the cursor is:

```
CLOSE cursorname
```

(the statement COMMIT that we will discuss further on has an implicit COMMIT for all cursors)

example

```
EXEC SQL
CLOSE CUR1
END-EXEC.
```

working with SELECT INTO
There is one exception from the above. That is the statement SELECT INTO. This statement aligns its functioning with the functioning of the host program in that it works on one row at a time. This is because this statement may only deliver one row from the database at a time. If this is not the case an error situation is the consequence.
The SELECT INTO statement has the following layout:

```
SELECT outputspecification
INTO :hostvariable1
      [, hostvaribele2]...
FROM table(-s)
[WHERE search condition]
```

© copyright Ton de Rooij

retrieved from www.tonderooij.com

example 1

```
EXEC SQL
SELECT      BNAME,
            BTHDATE,
            ADDRESS,
            TOWN
INTO   :BNAME,
       :BTHDATE,
       :ADDRESS,
       :TOWN
FROM BORROWER
WHERE BNAME = 'WOLFF'
END-EXEC.
```

example 2

```
EXEC SQL
SELECT      SUM(SALARY),
            MAX(SALARY),
            MIN(SALARY)
INTO        :TOTALSALARY,
            :HIGHESTSALARY,
            :LOWESTSALARY
FROM EMPLOYEE
WHERE LOCNAME = 'MADISON'
END-EXEC.
```

Transactions

what is a transaction

A transaction is a logical unit of work that is processed by a program. The amount of work must be chosen in a way that after the transaction the data in the database fulfil all integrity demands. A problem is that at the same time more than one transaction (by different programs) can be in execution. A transaction may as a consequence of this be reading 'in between' results of other transactions. This can be partially or totally avoided by isolating the transactions from other transactions. Isolating transactions from other transactions must be set by the program that executes them.
We will show how this must be done.

transactions in embedded SQL

In embedded SQL transactions can consist of more than one SQL statement. In order to process the transaction correctly it must be clear when a transaction starts and when the (same) transaction ends. Moreover it must be possible to undo a transaction. Finally transactions must not interfere with each other.

start and end of a transaction

A transaction starts with a random SQL statement and ends with the statement COMMIT. This may have as a consequence that a lot of SQL statements can be executed in one transaction. The syntax for ending a transaction is:

```
COMMIT
```

example:

```
EXEC SQL
COMMIT
END-EXEC.
```

to undo a transaction

A transaction can be undone by the statement:

```
ROLLBACK
```

This statement does undo the transaction that is being executed by the initiating program. Everything that the transaction has altered in the database is being undone. A side effect of the statement ROLLBACK is that all cursors are being closed.

example

```
EXEC SQL
ROLLBACK
END-EXEC.
```

programs that execute transactions at the same time

When we execute more than one transaction at the same time several problems can occur. These problems are:
- dirty read
- non-repeatable read
- phantoms

dirty read

Dirty read is the situation in which the transaction has read rows that have not existed or have not existed in the shape the transaction has read them.
This could happen as follows. Suppose transaction T1 updates a row or inserts a new row into the database. Suppose that transaction T2 reads this altered or new row. Suppose further that transaction T1 executes a ROLLBACK that undoes the changes in the row or deletes the row in which case the row has never existed. In that case transaction T2 has been working with a row in which there is a change that never has existed or with a row that never has existed.

non-repeatable read

Non-repeatable read is the situation in which the transaction is confronted with changes in rows the

© copyright Ton de Rooij

retrieved from www.tonderooij.com

transaction is working on.
This could happen as follows. Suppose transaction T1 reads a row. Suppose transaction T2 alters or deletes this row. Suppose transaction T1 tries to read the row again. Then transaction T1 becomes aware that the row is altered or deleted. This can occur, but transactions do not want to have changes in rows that they are working on.

phantoms
The phenomenon phantoms is the situation in which during the execution of a transaction all of a sudden rows appear that were not present earlier on in the transaction.
This could happen as follows. Suppose transaction T1 is working on a set of more than one rows. These rows have been read before transaction T2 adds a row that complies with the conditions with which transaction T1 collected its set of rows. Suppose transaction T1 wants to update the database based on the previously collected set of rows. It may very well be that the update is refused as a consequence of the newly added row. The reason for the refusal can be that for instance a CHECK constraint ascertains a violation that it would not have ascertained based on the previous set. In that case transaction T1 endures a phantom.

keeping transactions free of problems caused by simultaneously processed other transactions
For dealing with problems caused by the simultaneous use of the database by more than one program, two settings must be invoked:
- access mode
- isolation level

access mode
The access mode signifies whether a program is only allowed to read data in the database or that the program is also allowed to make changes in the content of the database. This is defined as follows:

```
SET TRANSACTION
            {READ ONLY | READ WRITE}
```

The access mode READ ONLY signifies that (with the exception of temporary tables) a program may only *read* data from the database.
The access mode READ WRITE signifies that a program is allowed to read data from the database as well as that it is allowed to make changes in the content of the database. READ WRITE is the default setting.

example

```
EXEC SQL
SET TRANSACTION READ ONLY
END-EXEC.
```

isolation level
The isolation level signifies the extent that transactions are allowed to cause one or more of the mentioned problems (dirty read, non-repeatable read, phantom read). During the writing of a program one needs to be aware of which problems can occur and which problems can't occur.
The program has to test whether problems occur. Possible isolation levels are:
- READ UNCOMMITTED; in case of READ UNCOMMITTED dirty read, repeatable read and phantom read problems can occur
- READ COMMITTED; in case of READ COMMITTED non repeatable read and phantom read problems can occur
- REPEATABLE READ; in case of REPEATABLE READ phantom read problems can occur
- SERIALIZABLE; in case of SERIALIZABLE none of the mentioned problems can occur (this setting consumes a lot of performance in order to meet these demands)

The isolation level can be set as follows:

```
SET TRANSACTION ISOLATION LEVEL
      { READ UCOMMITTED |
        READ COMMITTED  |
        REPEATABLE READ |
        SERIALIZABLE}
```

example

```
EXEC SQL
SET TRANSACTION ISOLATION LEVEL
                REPEATABLE READ
END-EXEC.
```

Error messages

During the execution of the SQL statements all kinds of things can go wrong. The host program must be able to be aware of this. Based on the kind of error it must be possible for the host program to steer the execution into an different direction. Two kinds of notification mechanisms can be used:

```
SQLERROR
SQLSTATE
```

© copyright Ton de Rooij

retrieved from www.tonderooij.com

SQLERROR
SQLERROR can be used by adding WHENEVER to the host program. We will show this by using COBOL as a host language. The syntax is:

```
WHENEVER SQLERROR GOTO :hostlabel
```

(the statement is part of the SQL statements)

example

```
EXEC SQL
WHENEVER SQLERROR
      GO TO :DEAL-WITH-ERROR
END-EXEC.
```

SQLSTATE
SQLSTATE can be used for all kinds of things. By the use of it, it can be established exactly what the error is that was found. The most important use however is establishing that no further data can be found. This gives code '02000'. SQLSTATE must be declared as a variable for the host program and initially be set at the value '00000'. SQLSTATE will get its values from the system.

example

```
PERFORM OPERATION1
      UNTIL SQLSTATE = '02000'.
```

Examples

We will repeat some of the examples that were given before. For the use of a cursor we will show how the statements correlate. The examples are:
- example UPDATE statement without the use of a cursor or host variables
- example DELETE statement with the use of a host variable
- example UPDATE statement with the use of a cursor
- example of the use of a cursor
- two examples of the use of SELECT INTO

example of an UPDATE statement

```
EXEC SQL
UPDATE LOCATION
SET CITY = 'WICKFORD'
WHERE LOCNAME = 'FASTBITE'
END-EXEC.
```

example of an DELETE statement with the use of a host variable

```
EXEC SQL
DELETE FROM LOCATION
WHERE LOCNAME = :LOCNAME
END-EXEC.
```

example of the UPDATE statement with the use of a cursor
(with the use of a cursor CUR3 that points one by one at the rows of the table EMPLOYEE. The employee gets a pay rise if the host program decides that the employee is entitled to it)

```
EXEC SQL
UPDATE EMPLOYEE
SET SALARY =
SALARY * (1+
  ( :RISEPERCENTATGE / 100))
WHERE CURRENT OF CUR3
END-EXEC.
```

example of the use of a cursor
The example below shows how parts of a COBOL program deal with awarding a higher salary to replacements of employees. The database used is the database 'labour division' of paragraph 6.2 of the book 'Databases and SQL'. Parts of the example were used in the previous explanations.

The part of the COBOL program looks as follows:

```
*REPLACEMENTS OF DIRECTORS GET THE
*SAME SALARY AS THE ONE THEY
*REPLACE.
   EXEC SQL
      DECLARE CUR1 CURSOR FOR
      SELECT      E1.ENR,
                  E1.SALARY,
                  E2.ENR,
                  E2.SALARY
      FROM  EMPLOYEE E1,
            EMPLOYEE E2,
            REPLACEMENT R
      WHERE E1.ENR  = R.REPLACED
      AND   R.SUBSTITUTION = E2.ENR
      AND   E1.FNAME = 'DIRECTOR'
      ORDER BY E1.ENAME
   END-EXEC.
   EXEC SQL
      WHENEVER SQLERROR
            GO TO :DEAL-WITH-ERROR
   END-EXEC.
   EXEC SQL
      OPEN CUR1
   END-EXEC.
      MOVE '00000' TO SQLSTATE.
   EXEC SQL
      FETCH CUR1
      INTO :REPLACED,
           :SALARY1,
           :SUBSTITUTE,
           :SALARY2
   END-EXEC.
```

© copyright Ton de Rooij

retrieved from www.tonderooij.com

```
IF SQLSTATE = '02000' GO TO FINISH.
PERFORM OPERATION1
        UNTIL SQLSTATE = '02000'.
EXEC SQL
   CLOSE CUR1
END-EXEC.
EXEC SQL
   COMMIT
END-EXEC.
   GO TO FINISH.
OPERATION1.
   MOVE SALARY1 TO SALARY2.
EXEC SQL
   UPDATE EMPLOYEE
   SET SALARY = :SALARY2
   WHERE ENR  = :SUBSTITUTE
END-EXEC.
EXEC SQL
   FETCH CUR1
   INTO :REPLACED,
        :SALARY1,
        :VERVANGER,
        :SALARY2
END-EXEC.
DEAL-WITH-ERROR.
DISPLAY
        "AN ERROR OCCURRED".
FINISH.
STOP RUN.
```

two examples of the use of SELECT INTO

Below are two examples that were previously used as an example of the use of 'SELECT INTO'. Notice how it is assured that in the output only one row can occur.

example 1

```
EXEC SQL
SELECT      BNAME,
            BTHDATE,
            ADDRESS,
            TOWN
INTO  :BNAME,
      :BTHDATE,
      :ADDRESS,
      :TOWN
FROM BORROWER
WHERE BNAME = 'WOLFF'
END-EXEC.
```

example 2

```
EXEC SQL
SELECT      SUM(SALARY),
            MAX(SALARY),
            MIN(SALARY)
INTO        :TOTALSALARY,
            :HIGHESTSALARY,
            :LOWESTSALARY
FROM EMPLOYEE
WHERE LOCNAME = 'MADISON'
END-EXEC.
```

© copyright Ton de Rooij

retrieved from www.tonderooij.com