R:BASE Cursors Explained

R:BASE Cursors Explained

by R:BASE Technologies, Inc.

An R:BASE cursor is a valuable programming control structure that enables traversal reference over the records in a database. It is a pointer to rows in a table that can step through rows one by one, performing the same action on each row. A cursor can be set to point to all the rows in a table or to a subset of rows.

R:BASE Cursors Explained

Printed: April 2017 in Murrysville, PA

First Edition

Table of Contents

1 R:BASE Cursors

A cursor is a valuable programming tool. It is a pointer to rows in a table. A cursor lets you step through rows one by one, performing the same action on each row. You can set a cursor to point to all the rows in a table or to a subset of rows. A cursor is set using the DECLARE CURSOR command.

The DECLARE CURSOR command does not work by itself, but is really a sequence of commands. In addition to the DECLARE CURSOR, the OPEN and FETCH commands are required. A WHILE loop is used to step through the rows and perform the programmed action on each row. The CLOSE or DROP command is used after the cursor has stepped through all the rows.

The basic sequence of commands for a cursor is as follows:

```
SET VAR vCustID INTEGER = NULL
SET VAR vCompany TEXT = NULL
DECLARE c1 CURSOR FOR +
   SELECT CustID, Company FROM Customer
OPEN c1
FETCH c1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2
WHILE SQLCODE <> 100 THEN
   -- Place code for row by row actions here.
   FETCH c1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2
ENDWHILE
DROP CURSOR c1
```

The DECLARE CURSOR command names the cursor and defines the set of rows. The cursor name is then used in the OPEN, FETCH, CLOSE, and DROP commands that reference it. A cursor name can be up to 18 characters long and follows the same naming conventions as all other names in R:BASE.

More than one cursor can be defined and open at a time. SELECT is used in the DECLARE CURSOR to identify the rows to step through. The SELECT part of a cursor declaration can point to rows from a single table or from multiple tables, and can choose all or only some of the columns from a table. You can use the GROUP BY clause as well as the WHERE and ORDER BY clauses of SELECT.

The OPEN command initializes the cursor and tells R:BASE you are ready to retrieve a row of data from the cursor. The OPEN command positions the cursor at the first row of the set of data defined by the SELECT in the cursor declaration.

The FETCH command retrieves a row of data into the specified variables. The number of variables must match the number of columns listed in the SELECT part of the DECLARE CURSOR command. Each variable has a corresponding indicator variable, which tells if a NULL value was retrieved. The list of variable pairs - data variable and indicator variable - is separated by commas.

The FETCH command sets SQLCODE, the SQL error variable. If a row was retrieved, SQLCODE is set to 0. After the last row is retrieved, FETCH sets SQLCODE to 100 - no more data. Using SQLCODE as the condition for the WHILE loop lets you easily retrieve and act on each successive row. Placing a second FETCH command immediately before the ENDWHILE command keeps fetching rows until the end of data is reached. Then the loop exits.

Within the WHILE loop, place whatever commands are needed to operate on each row. You can look up additional data, perform mathematical calculations, update data, and so on.

When the cursor completes and the WHILE loop is exited, the cursor is dropped with the DROP CURSOR command. A cursor name must be dropped before it can be declared again. DROP removes a cursor definition from memory; to use the cursor again, it must be declared with the

DECLARE CURSOR command. CLOSE leaves a cursor definition in memory; to use the cursor again, it is opened with the OPEN command. After a cursor has been closed, an OPEN repositions the pointer at the first row of the cursor definition. CLOSE is most often used with nested cursors, DROP with individual cursors.

When a cursor is open, you can use a special WHERE clause option, WHERE CURRENT OF cursorname. This WHERE clause works with the UPDATE, DELETE, and SELECT commands to perform the specified action on the row the cursor is currently pointing at. The DELETE deletes the entire row; the SELECT, and UPDATE only operate on columns included in the SELECT part of the DECLARE CURSOR command. Note that not every cursor definition supports use of the WHERE CURRENT OF cursorname.

It is not required to use the WHERE CURRENT OF cursorname in your WHERE clause. A WHERE clause that explicitly points to a row of data using values stored in variables can be used. The unique row identifier is fetched into a variable, then that value is used to access rows in the cursor table or other tables.

```
-- The special WHERE clause WHERE CURRENT OF
-- points to the current row of the cursor.
SELECT CustID, Company FROM Customer +
WHERE CURRENT OF c1
UPDATE Customer SET CustID = (CustID + 1000) +
   WHERE CURRENT OF c1
DELETE FROM Customer WHERE CURRENT OF c1

-- Alternatively, use an explicit WHERE
-- clause to access a row.

SELECT CustID, Company FROM Customer +
   WHERE CustID = .vCustID
UPDATE Customer SET CustID = (CustID + 1000) +
   WHERE CustID = .vCustID
DELETE FROM Customer WHERE CustID = .vCustID
```

This is the basic cursor structure. Other types of cursors and cursor structures that are used are: multi-table cursors, non-updateable cursors, nested cursors, resettable cursors, and scrolling cursors. Each is described below.

2 Multi-Table Cursors

A multi-table cursor includes more than one table in the SELECT part of the cursor declaration. The tables can be linked directly within the DECLARE CURSOR command; avoiding steps to define a view to retrieve data from more than one table.

The DECLARE CURSOR command has the full capabilities of the SELECT command to do multi-table queries. As with the SELECT command itself, you list the columns to retrieve, the tables to get the data from, then link the tables in the WHERE clause. For example,

```
SET VAR vCustID INTEGER = NULL
SET VAR vCompany TEXT = NULL
SET VAR vTransID INTEGER = NULL
SET VAR vTransDate DATE = NULL
SET VAR vInvoiceTotal CURRENCY = NULL

-- Check for an existing cursor, and drop it if exists
SET VAR vCheckCursor INTEGER = (CHKCUR('C1'))
IF vCheckCursor = 1 THEN
  DROP CURSOR C1
ENDIF

-- Select data from both the Customer and Transmaster tables.
DECLARE C1 CURSOR FOR SELECT +
CustID, Company, TransID, TransDate, InvoiceTotal +
   FROM Customer, TransMaster +
   WHERE Customer.CustID = TransMaster.CustID
OPEN C1

-- The fetch retrieves all the specified columns into variables.
FETCH C1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2 +
   vTransID INDICATOR iv3, vTransDate INDICATOR iv4, +
   vInvoiceTotal INDICATOR iv5
WHILE SQLCODE <> 100 THEN

   -- Place code for row by row actions here.
   -- An explicit WHERE clause must be used,
   -- WHERE CURRENT OF is not supported with
   -- multi-table cursors.

   -- Get the next row
   FETCH C1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2 +
      vTransID INDICATOR iv3, vTransDate INDICATOR iv4, +
      vInvoiceTotal INDICATOR iv5
ENDWHILE
DROP CURSOR C1
CLEAR VAR vCustID, vCompany, vTransID, vTransDate, vInvoiceTotal
RETURN
```

Notice that the basic structure of the cursor commands doesn't change. You still declare the cursor, open it, fetch the first row, then use a WHILE loop to step through each row. There is no limit to the number of tables that can be included in a DECLARE CURSOR command. The tables are joined together in the same way they are joined with a regular SELECT command.

A multi-table cursor definition is a non-updateable cursor, however. You cannot update the cursor directly by using WHERE CURRENT OF cursorname. You must use explicit WHERE clauses to access the cursor tables.

3 Non-Updateable Cursors

A non-updateable cursor is one that does not support use of the special WHERE clause WHERE CURRENT OF cursorname. An explicit WHERE clause must be used to access data in the tables.

A non-updateable cursor is a multi-table cursor, or a cursor that is defined, for example, using the GROUP BY clause. The SELECT command that defines the cursor rows does not allow the cursor to point back to a single specific row in a table.

Non-updateable cursors are a very useful part of the DECLARE CURSOR structure. Use the power of the SELECT command in the DECLARE CURSOR declaration to dramatically improve the performance of a cursor. The more work the cursor does, the less your program has to do and the faster and more efficiently it will run.

When using a non-updateable cursor, make sure you fetch a unique row identifier for use in WHERE clauses.

4 Nested Cursors

A nested cursor involves two DECLARE CURSOR definitions. The second cursor is dependent on the first and its cursor definition uses a variable value fetched by the first cursor.

There is a specific structure recommended for nested cursors - a row is retrieved from cursor one, then the matching rows in cursor two are retrieved and stepped through. Then the next row is retrieved from cursor one and its matching rows from cursor two are stepped through. The process continues until all rows have been retrieved from cursor one.

Example

```
SET VAR vCustID INTEGER = NULL
SET VAR vCustID1 INTEGER = NULL
SET VAR vCompany TEXT = NULL
SET VAR vFirstName TEXT = NULL
SET VAR vLastName TEXT = NULL

-- Check for existing cursors, and drop it if exists
SET VAR vCheckCursor INTEGER = (CHKCUR('c1'))
IF vCheckCursor = 1 THEN
  DROP CURSOR c1
ENDIF
SET VAR vCheckCursor INTEGER = (CHKCUR('c2'))
IF vCheckCursor = 1 THEN
  DROP CURSOR c2
ENDIF

-- The DECLARE commands are done together
-- at the top of the program.
-- An OPEN cursor does not need to immediately
-- follow the corresponding DECLARE CURSOR command
DECLARE c1 CURSOR FOR SELECT CustID, Company +
   FROM Customer ORDER BY Company
-- The second cursor uses a variable in the
-- WHERE clause. This variable, vCustID, must be
-- defined earlier in the program.
-- The cursor retrieves rows for a single Customer only
DECLARE c2 CURSOR FOR +
   SELECT CustID, ContFName, ContLName +
   FROM Contact WHERE CustID = .vCustID

-- Cursor c1 is opened and the first row retrieved
-- from the Customer table
OPEN c1
FETCH c1 INTO vCustID1 INDICATOR iv1, vCompany INDICATOR iv2
WHILE SQLCODE <> 100 THEN

   -- Cursor c2 is opened, it points to all the
   -- rows in the Contact table that match the
   -- CustID fetched into vCustID by cursor c1.
   OPEN c2

   -- Get the first row from the contact table and step
   -- through all matching rows.
   FETCH c2 INTO vCustID1 INDICATOR iv1, vFirstName INDICATOR iv2, +
      vLastName INDICATOR iv3
      WHILE SQLCODE <> 100 THEN

         -- Place code here to do row by row actions

         --Get the next row for cursor c2
         FETCH c2 INTO vCustID1 INDICATOR iv1, vFirstName INDICATOR iv2, +
            vLastName INDICATOR iv3
      ENDWHILE

      -- After all the matching rows in the contact table
      -- have been processed, close cursor c2 and get the
      -- next row from the Customer table.
      -- Cursor c2 is closed and not dropped because
      -- the definition will be reused for the next
      -- row from cursor c1.
      CLOSE c2

      -- Get the next row for cursor c1
      FETCH c1 INTO vCustID1 INDICATOR iv1, vCompany INDICATOR iv2
   ENDWHILE

   -- Both cursors are dropped when all the rows
   -- in the Customer table have been retrieved.
   DROP CURSOR c2
   DROP CURSOR c1
   CLEAR VAR vCustID, vCustID1, vCompany, vFirstName, vLastName
   RETURN
```

You can use the same WHILE loop condition, SQLCODE <> 100, for both cursors. This works very well and there is no conflict between the two loops. The relative FETCH command sets the value of SQLCODE. Notice that the FETCH from cursor c2 is right before the ENDWHILE of the inner WHILE loop ensuring that that FETCH command is the one being tested by the WHILE loop. The FETCH from cursor c1 is right before the ENDWHILE of the outer WHILE loop, which then continues based on cursor c1. This placement of the DECLARE, OPEN, FETCH, WHILE, and ENDWHILE statements will always work. Just make sure the ENDWHILE is the next command after the FETCH.

With nested cursors, the inner cursor is closed and opened so that it always references the matching rows from the outer cursor. An alternative to opening and closing the inner cursor is to use the RESET option on the OPEN command.

5 Resettable Cursors

A DECLARE CURSOR can use a variable in its WHERE clause. Each time the cursor is opened, the WHERE clause is reevaluated using the current variable value and identifies a new set of data.

You can CLOSE and OPEN a defined cursor, or use the OPEN cursorname RESET command. Don't use the CLOSE command if you place the RESET option on the OPEN command. The RESET option automatically reevaluates the variable value and identifies a new set of data for the cursor.

OPEN cursorname RESET is commonly used with nested cursors. The second cursor is dependent on a variable fetched by the first cursor. By using RESET, you won't need to CLOSE the inner cursor each time.

Using the RESET option on OPEN is faster using than the OPEN, CLOSE sequence of commands.

6 Scrolling Cursors

Normally, cursors move through the data in one direction only, from top to bottom. They move forward one-by-one through the set of defined rows. Once a row has been accessed and passed over, you can't get back to it. The rows can be ordered in the cursor definition - the top to bottom order is not necessarily the table order.

When a cursor is defined as a scrolling cursor, you gain the capability of moving both forwards and backwards through the rows of data and can also jump past rows.

To define a cursor as a scrolling cursor, include the word SCROLL in the DECLARE CURSOR command. For example,

```
DECLARE c1 SCROLL CURSOR FOR SELECT ...
```

The word SCROLL comes right after the cursor name. If SCROLL is not included in the cursor definition, the cursor can only move forward through the rows one at a time.

Once a cursor is defined as a scrolling cursor, a number of additional options on the FETCH command become available. These options are as follows; note that the directions and positions are based on the order of the rows as specified by the DECLARE CURSOR command, not on the order of the rows in the actual table:

NEXT - The default option if none is specified on the FETCH command. NEXT moves the cursor forward through the rows, it gets the next available row based on the current cursor position. NEXT steps through the rows one-by-one going forward.

PRIOR - Moves the cursor backwards through the rows. The PRIOR option gets the previous row based on the current cursor position, and steps through the rows one-by-one going backwards.

FIRST - Moves the cursor from its current position to the first row. This option jumps immediately to the first row as determined by the DECLARE CURSOR command. A FETCH NEXT then finds the second row. The cursor is repositioned at the beginning of the set of rows.

LAST - Moves the cursor from its current position immediately to the last row as specified by the DECLARE CURSOR command. A FETCH PRIOR then finds the next to last row; a FETCH NEXT returns "end of data encountered". LAST jumps over the rows between the current cursor position and the last row.

ABSOLUTE n - Moves the cursor the specified number of rows from the first row of data as determined by the DECLARE CURSOR and OPEN commands. A positive number must be specified; you can't use this option to move backwards. The intervening rows are jumped over. You can't jump past the last row; if the number given is greater than the number of rows retrieved, an "end of data" error is returned.

RELATIVE n - Moves the cursor the specified number of rows from the current cursor position. This option moves the cursor either forwards or backwards - forwards if a positive number is specified, backwards if a negative number is specified. The intervening rows are jumped over. You can't jump past the last row or the first row; an "end of data" error is returned if the specified number would take you past the beginning or end of the selected rows.

Example
To see how a scrolling cursor can be used in an application, imagine you have a group of customers to contact each day. The scrolling cursor retrieves the list of customers for today. They are ordered by Company name. The first row is brought up in a menuless form. The form remains on the screen when you are done with the record, and a CHOOSE menu pops up giving the user choices as to which record to select next.

You can: move through the list of customers one-by-one, both forwards and backwards, jump to the last record and back to the first record, jump past a group of records, and search for a particular record by last name or by Company name

Each time you select a record, the cursor is repositioned ready for the next selection.

```
--WALKLIST.RMD
--scroll through a list of customers
SET MESSAGES OFF
SET ERROR MESSAGES OFF

SET VAR vCustID INTEGER = NULL
SET VAR vLastName TEXT = NULL
SET VAR vCompany TEXT = NULL
SET VAR vNum INTEGER = NULL
SET VAR vPlus INTEGER = NULL
SET VAR vMinus INTEGER = NULL
SET VAR vSearch TEXT = NULL

SET VAR vCheckCursor INTEGER = (CHKCUR('C1'))
IF vCheckCursor = 1 THEN
  DROP CURSOR C1
ENDIF

--Define the scrolling cursor
DECLARE C1 scroll CURSOR FOR +
   SELECT CustId, LastName, Company FROM Customer +
   WHERE CallDate = .#DATE ORDER BY Company

--Open the cursor and get the first row
OPEN C1
FETCH FIRST FROM C1 INTO +
   vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
   vCompany INDICATOR iv3
WHILE SQLCODE <> 100 THEN

   --Bring up the form with the data from the first row.
   --After the form is closed, choose from the menu which record to retrieve next
   EDIT USING CustForm WHERE CustId = .vCustId
   CHOOSE vAction FROM #LIST +
      'Next Customer,Previous Customer,Jump Forward "n",Jump Backward "n",+
      Last Customer,First Customer,Search by Last Name,Search by Company' +
      TITLE 'Select Customer' CAPTION 'Choose' LINES 8 FORMATTED
   IF vAction = '[Esc]' THEN
      RETURN
   ENDIF

   --The switch/case block determines which record to retrieve
   SWITCH (.vAction)

   --Move forward one row at a time
   CASE 'Next Customer'
      FETCH NEXT FROM C1 INTO +
         vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
         vCompany INDICATOR iv3

      --If already on the last row, stay there
      IF SQLCODE = 100 THEN
```

```
        FETCH LAST FROM C1 INTO +
          vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
          vCompany INDICATOR iv3
      ENDIF
      BREAK

    --Move backward one row at a time
    CASE 'Previous Customer'
      FETCH PRIOR FROM C1 INTO +
        vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
        vCompany INDICATOR iv3

      --If already on the first row, stay there
      IF SQLCODE = 100 THEN
        FETCH FIRST FROM C1 INTO +
          vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
          vCompany INDICATOR iv3
      ENDIF
      BREAK

    --Move forward the specified number of records
    CASE 'Jump Forward "n"'
      DIALOG 'How many to jump forward?' vNum=4 vEndKey 1
      SET VAR vPlus = (INT(.vNum))

      --R:BASE counts from the current cursor position
      FETCH RELATIVE .vPlus FROM C1 INTO +
        vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
        vCompany INDICATOR iv3

      --If the number of records to jump past takes you beyond the last
      --record, the last record is retrieved
      IF SQLCODE = 100 THEN
        FETCH LAST FROM C1 INTO +
          vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
          vCompany INDICATOR iv3
      ENDIF
      BREAK

    --Move backward the specified number of records
    CASE 'Jump Backward "n"'
      DIALOG 'How many to jump backward?' vNum=4 vEndKey 1
      SET VAR vMinus = (INT(.vNum) * -1)

      --R:BASE counts from the current cursor position
      FETCH RELATIVE .vMinus FROM C1 INTO +
        vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
        vCompany INDICATOR iv3

      --If the number of records to jump past takes you beyond the first
      --record, the first record is retrieved
      IF SQLCODE = 100 THEN
        FETCH FIRST FROM C1 INTO +
          VCustId INDICATOR ICustId, VLastname INDICATOR ILastname, +
          VCompany INDICATOR ICompany
      ENDIF
      BREAK
```

```
--Jump to the last record Next Customer from the last record
--returns end-of-data
CASE 'Last Customer'
   FETCH LAST FROM C1 INTO +
      vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
      vCompany INDICATOR iv3
   BREAK

--Jump to the first record Prior Customer from the first record
--returns end-of-data
CASE 'First Customer'
   FETCH FIRST FROM C1 INTO +
      vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
      vCompany INDICATOR iv3
   BREAK

--Prompt for the last name to find
CASE 'Search by Last Name'
   DIALOG 'Enter the last name to find' vSearch vEndKey 1
   IF vEndKey = '[Esc]' THEN
      BREAK
   ENDIF

   WHILE #PI <> 0.0 THEN

      --Search forward for a matching record
      FETCH NEXT FROM C1 INTO +
         vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
         vCompany INDICATOR iv3

      --If a match is found, the row is displayed and the cursor
      --repositioned at that row
      IF vLastName CONTAINS .vSearch THEN
         BREAK
      ENDIF

      --If no match was found, the search can be continued from the first row.
      IF SQLCODE = 100 THEN
         DIALOG 'No match found. Continue search from beginning?' +
            vResponse vEndKey YES
         IF vEndKey = '[Esc]' THEN
            BREAK
         ENDIF

         IF vResponse = 'YES' THEN
            FETCH FIRST FROM C1 INTO +
               vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
               vCompany INDICATOR iv3
            IF vLastName CONTAINS .vSearch THEN
               BREAK
            ENDIF
         ELSE
            --If the search is not continued, the last row is retrieved
            FETCH LAST FROM C1 INTO +
               vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
               vCompany INDICATOR iv3
               BREAK
            ENDIF
         ENDIF
      ENDWHILE
      BREAK

   --Prompt for the Company name to find
   CASE 'Search by Company'
      SET VAR vSearch = NULL
      DIALOG 'Enter the Company to find' vSearch vEndKey 1
      IF vEndKey = '[Esc]' THEN
         BREAK
      ENDIF

      --Search forward for a matching Company record If a match is found,
      --the row is displayed and the cursor repositioned at that row
      WHILE #PI <> 0.0 THEN
         FETCH NEXT FROM C1 INTO +
            vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
            vCompany INDICATOR iv3
         IF vCompany CONTAINS .vSearch THEN
            BREAK
         ENDIF

         --If no match was found, the search can be continued from the first row.
         IF SQLCODE = 100 THEN
            DIALOG 'No match found. Continue search from beginning?' +
               vResponse vEndKey YES
            IF vEndKey = '[Esc]' THEN
               BREAK
            ENDIF
               IF vResponse = 'YES' THEN
               FETCH FIRST FROM C1 INTO +
                  vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
                  vCompany INDICATOR iv3
               IF vCompany CONTAINS .vSearch THEN
                  BREAK
               ENDIF
            ELSE
               --If the search is not continued, the last row is retrieved.
               FETCH LAST FROM C1 INTO +
                  vCustId INDICATOR iv1, vLastName INDICATOR iv2, +
                  vCompany INDICATOR iv3
               BREAK
            ENDIF
         ENDIF
      ENDWHILE
      BREAK
   ENDSW
ENDWHILE
DROP CURSOR C1
CLEAR VAR vCustID, vLastName, vCompany, vNum, vPlus, vMinus, vSearch
RETURN
```

7 Optimizing Cursors

DECLARE CURSOR is not always the fastest way to accomplish a task, particularly an UPDATE or an INSERT. If you can replace your DECLARE CURSOR routine with a single SQL command, you will dramatically improve performance. However, some tasks require a DECLARE CURSOR.

Let the cursor do the work
To improve the performance of a DECLARE CURSOR routine, do as much work in the DECLARE CURSOR as possible. This is the single most important factor in improving cursor performance. Do whatever work can be done in the SELECT command part of the DECLARE CURSOR - select as many columns of data as possible and also do calculations there if you can. The DECLARE CURSOR does the operation only once; inside the WHILE loop, the command is repeated for each row that is stepped through.

To do actions for unique rows only, use SELECT DISTINCT in the cursor definition instead of adding code to your WHILE loop to test the row values to see if they are the same or different. Use the SELECT functions to sum, average, count and so on in the cursor definition instead of for each row in the WHILE loop. Select as many columns as possible in the DECLARE CURSOR rather than retrieve the data each row in the WHILE loop.

The fewer commands repeated in the WHILE loop, the faster your DECLARE CURSOR will run. Remember that each command in the WHILE loop is repeated for each row retrieved by the DECLARE CURSOR. Use optimized variables in the WHILE loop -initialize each variable outside the WHILE loop, and do not change the data type of variables in the loop.

Following are two examples showing progressive changes made to a DECLARE CURSOR routine to improve performance.

7.1 Example 1

The task is to sum the extended price column in the transaction detail, Transdetail, table for each transaction ID, then update the transaction header, Transmaster, table with the sum. An initial approach is to declare a cursor on the header table, then step through all matching rows in the detail table. After all the matching detail rows have been processed, the header table is updated.

```
*(Post1.RMD -- the worst case)
-- nested declare cursors
-- strictly linear programming
SET VAR vTransID INTEGER = NULL
SET VAR vNetAmount CURRENCY = NULL
SET VAR vTotal CURRENCY = NULL
SET VAR vPrice CURRENCY = NULL

SET VAR vCheckCursor INTEGER = (CHKCUR('c1'))
IF vCheckCursor = 1 THEN
  DROP CURSOR c1
ENDIF
SET VAR vCheckCursor INTEGER = (CHKCUR('c2'))
IF vCheckCursor = 1 THEN
  DROP CURSOR c2
ENDIF

DECLARE c1 CURSOR FOR SELECT TransID, NetAmount +
   FROM TransMaster
OPEN c1
FETCH c1 INTO vTransID INDICATOR iv1, vNetAmount INDICATOR iv2
```

```
WHILE SQLCODE <> 100 THEN
   DECLARE c2 CURSOR FOR SELECT ExtPrice +
      FROM TransDetail WHERE TransID = .vTransID
   OPEN c2
   FETCH c2 INTO vPrice INDICATOR iv3
   WHILE SQLCODE <> 100 THEN
      SET VAR vTotal = (.vTotal + .vPrice)
      FETCH c2 INTO vPrice INDICATOR iv3
   ENDWHILE
   DROP CURSOR c2
   UPDATE TransMaster SET NetAmount = .vTotal +
      WHERE CURRENT OF c1
   SET VAR vTotal = NULL
   FETCH c1 INTO vTransID INDICATOR iv1, vNetAmount INDICATOR iv2
ENDWHILE
DROP CURSOR c1
CLEAR VAR vTransID, vNetAmount, vTotal, vPrice
RETURN
```

We can speed up this code by following the recommended structure for nested cursors. If we move the second DECLARE CURSOR out of the WHILE loop and reset the cursor instead of dropping it, this command file will execute faster. However, the best way to improve this code is by removing the second DECLARE CURSOR altogether. We don't need to step through all the rows in the detail table - we can compute the sum with a single SELECT command.

```
*(Post2.RMD - a little bit better)
-- use the SELECT or COMPUTE command
-- to calculate the sum instead of a nested cursor
SET VAR vTransID INTEGER = NULL
SET VAR vNetAmount CURRENCY = NULL
SET VAR vPrice CURRENCY = NULL

SET VAR vCheckCursor INTEGER = (CHKCUR('c1'))
IF vCheckCursor = 1 THEN
  DROP CURSOR c1
ENDIF

DECLARE c1 CURSOR FOR SELECT TransID, NetAmount +
   FROM TransMaster
OPEN c1
FETCH c1 INTO vTransID INDICATOR iv1, vAmount INDICATOR iv2
WHILE SQLCODE <> 100 THEN
   SELECT SUM(ExtPrice) INTO vPrice +
      FROM TransDetail WHERE TransID = .vTransID
   -- if no matching rows in the Transdetail table,
   -- vPrice is null
   IF vPrice IS NOT NULL THEN
      UPDATE TransMaster SET NetAmount = .vPrice +
         WHERE CURRENT OF c1
   ENDIF
   SET VAR vPrice = NULL
   FETCH c1 INTO vTransID INDICATOR iv1, vAmount INDICATOR iv2
ENDWHILE
DROP CURSOR c1
CLEAR VAR vTransID, vNetAmount, vPrice
RETURN
```

This simple change reduced the number of commands in the program, which in turn improved performance. All the commands inside the WHILE loop still need to be executed for as many rows as are in the Transmaster table, however. The Transmaster table has fewer rows than the Transdetail table, so a valid assumption is to place the cursor on the Transmaster table to repeat the WHILE loop the fewest times.

However, if we place the cursor on the detail table instead of on the header table, the sum can be calculated directly in the DECLARE CURSOR. Because the command is grouped by the transaction ID, the same number of rows is retrieved by the cursor. The only commands to repeat in the WHILE loop are the UPDATE and the FETCH to get the next row. At first this might seem backwards, but computing the sum in the DECLARE CURSOR is much faster.

```
*(Post3.RMD - better yet)
-- declare the cursor on the detail table and
-- do the sum directly in the cursor definition
SET VAR vTransID INTEGER = NULL
SET VAR vNetAmount CURRENCY = NULL
SET VAR vPrice CURRENCY = NULL

SET VAR vCheckCursor INTEGER = (CHKCUR('c1'))
IF vCheckCursor = 1 THEN
  DROP CURSOR c1
ENDIF

DECLARE c1 CURSOR FOR SELECT TransID, SUM(ExtPrice) +
   FROM TransDetail GROUP BY TransID
OPEN c1
FETCH c1 INTO vTransID INDICATOR iv1, vPrice INDICATOR iv2
WHILE SQLCODE <> 100 THEN
   -- this is a non-updateable cursor so an explicit
   -- WHERE clause is used
   UPDATE TransMaster SET NetAmount = .vPrice +
      WHERE TransID = .vTransID
   FETCH c1 INTO vTransID INDICATOR iv1, vPrice INDICATOR iv2
ENDWHILE
DROP CURSOR c1
CLEAR VAR vTransID, vNetAmount, vPrice
RETURN
```

The number of commands has been reduced by over half from the first program, and performance by more than that. The multi-table update command is actually the fastest way to accomplish this task.

```
*(Post4.RMD - do a multi-table update if you can)
-- multi table update command, a view is used
-- to first calculate the sum and create a
-- one-one relationship
DROP VIEW TransView
CREATE VIEW TransView (TransID, Amount) AS +
   SELECT TransID, SUM(ExtPrice) +
   FROM TransDetail GROUP BY TransID
UPDATE TransMaster SET NetAmount = Amount +
   FROM TransMaster ,TransView t2 +
   WHERE TransMaster.TransID = t2.TransID
```

7.2 Example 2

The task here is to create a quick report of companies from the Customer table and their corresponding contact names from the Contact table. Using nested cursors makes printing the Company information once followed by the many rows of contact information easier.

```
*(CustRep1.RMD - the worst case)
-- nested cursors are used with the declare for
-- the second cursor inside the while loop of
-- the first cursor. Also, the data is retrieved
-- with a SELECT command instead of in the
-- cursor definition
SET VAR vCustID INTEGER = NULL
SET VAR vCompany TEXT = NULL
SET VAR vAddress TEXT = NULL
SET VAR vCity TEXT = NULL
SET VAR vState TEXT = NULL
SET VAR vZipCode TEXT = NULL
SET VAR vPhone TEXT = NULL
SET VAR vCityStateZip TEXT = NULL
SET VAR vFName TEXT = NULL
SET VAR vLName TEXT = NULL
SET VAR vFullName TEXT = NULL

SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
IF vCheckCursor1 = 1 THEN
  DROP CURSOR c1
ENDIF
SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
IF vCheckCursor2 = 1 THEN
  DROP CURSOR c2
ENDIF

-- Only the unique row identifier is specified in
-- the cursor definition
DECLARE c1 CURSOR FOR SELECT CustID FROM Customer +
   ORDER BY CustID
OPEN c1
FETCH c1 INTO vCustID INDICATOR iv1
WHILE SQLCODE <> 100 THEN

   -- Retrieve and display the rest of the
   -- data for a Customer
   SELECT Company, CustAddress, CustCity, +
      CustState, CustZip, CustPhone INTO +
      vCompany INDICATOR iv1, vAddress INDICATOR iv2, +
      vCity INDICATOR iv3, vState INDICATOR iv4, +
      vZipCode INDICATOR iv5, vPhone INDICATOR iv6 +
      FROM Customer WHERE CustID = .vCustID
   SET VAR vCityStateZip = (.vCity + ',' & .vState & .vZipCode)
   WRITE .vCustID, .vCompany
   WRITE .vAddress
   WRITE .vCityStateZip

   -- Declare a cursor to identify matching contact rows
   DECLARE c2 CURSOR FOR SELECT  ContFName, ContLName +
      FROM Contact WHERE CustID = .vCustID
   OPEN c2
   FETCH c2 INTO vFName INDICATOR iv1, vLName INDICATOR iv2
   WHILE SQLCODE <> 100 THEN
      SET VAR vFullName = (.vFName & .vLName)
      WRITE .vFullName
      FETCH c2 INTO vFName INDICATOR iv1, vLName INDICATOR iv2
   ENDWHILE
   DROP CURSOR c2
   FETCH c1 INTO vCustID INDICATOR iv1
ENDWHILE
DROP CURSOR c1
CLEAR VAR vCustID, vCompany, vAddress, vCity, vState, +
   vZipCode, vPhone, vCityStateZip, vFName, vLName, vFullName
RETURN
```

The next code segment shows the recommended structure for nested cursors. The second DECLARE CURSOR is moved to the top of the program, and the second cursor is opened and closed, not declared and dropped. Just this simple change improves performance.

```
   *(CustRep2.RMD - move cursor out of WHILE loop)
   SET VAR vCustID INTEGER = NULL
   SET VAR vCompany TEXT = NULL
   SET VAR vAddress TEXT = NULL
   SET VAR vCity TEXT = NULL
   SET VAR vState TEXT = NULL
   SET VAR vZipCode TEXT = NULL
   SET VAR vPhone TEXT = NULL
   SET VAR vCityStateZip TEXT = NULL
   SET VAR vFName TEXT = NULL
   SET VAR vLName TEXT = NULL
   SET VAR vFullName TEXT = NULL

   SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
   IF vCheckCursor1 = 1 THEN
     DROP CURSOR c1
   ENDIF
   SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
   IF vCheckCursor2 = 1 THEN
     DROP CURSOR c2
   ENDIF

   DECLARE c1 CURSOR FOR SELECT CustID +
      FROM Customer ORDER BY CustID
   DECLARE c2 CURSOR FOR SELECT ContFName, contlname +
      FROM contact WHERE CustID = .vCustID

   -- Get the first row of data for a Customer
   OPEN c1
   FETCH c1 INTO vCustID INDICATOR iv1
   WHILE SQLCODE <> 100 THEN

      -- Retrieve and display the rest of the
      -- data for a Customer
      SELECT Company, CustAddress, CustCity, +
         CustState, CustZip, CustPhone INTO +
         vCompany INDICATOR vi1, vAddress INDICATOR vi2, +
         vCity INDICATOR vi3, vState INDICATOR vi4, +
         vZipCode INDICATOR vi5, vPhone INDICATOR vi6 +
         FROM Customer WHERE CustID = .vCustID
      SET VAR vCityStateZip = (.vCity + ',' & .vState & .vZipCode)
      WRITE .vCustID, .vCompany
      WRITE .vAddress
      WRITE .vCityStateZip

      -- Open cursor c2, retrieve and display
      -- the matching contact data
      OPEN c2
      FETCH c2 INTO vFName INDICATOR i1, vLName INDICATOR i2
      WHILE SQLCODE <> 100 THEN
         SET VAR vFullName = (.vFName & .vLName)
         WRITE .vFullName
         FETCH c2 INTO vFName INDICATOR i1, vLName INDICATOR i2
      ENDWHILE

      -- Close cursor c2 and get the next row of
      -- Customer data
      CLOSE c2
      FETCH c1 INTO vCustID INDICATOR iv1
   ENDWHILE
   DROP CURSOR c1
   DROP CURSOR c2
   CLEAR VAR vCustID, vCompany, vAddress, vCity, vState, +
      vZipCode, vPhone, vCityStateZip, vFName, vLName, vFullName
   RETURN
```

Moving the data retrieval to the DECLARE CURSOR command instead of using a separate SELECT command again improves performance.

```
   *(CustRep3.RMD)
   --retrieve data through DECLARE CURSOR
   SET VAR vCustID INTEGER = NULL
   SET VAR vCompany TEXT = NULL
   SET VAR vAddress TEXT = NULL
   SET VAR vCity TEXT = NULL
   SET VAR vState TEXT = NULL
   SET VAR vZipCode TEXT = NULL
   SET VAR vPhone TEXT = NULL
   SET VAR vCityStateZip TEXT = NULL
   SET VAR vFName TEXT = NULL
   SET VAR vLName TEXT = NULL
   SET VAR vFullName TEXT = NULL

   SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
   IF vCheckCursor1 = 1 THEN
     DROP CURSOR c1
   ENDIF
   SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
   IF vCheckCursor2 = 1 THEN
     DROP CURSOR c2
   ENDIF

   -- retrieve all the data through the DECLARE CURSOR
   -- command instead of SELECT
   SET VAR vCustID INTEGER
```

```
DECLARE c1 CURSOR FOR SELECT CustID, Company, +
   CustAddress, CustCity, CustState, CustZip, +
   CustPhone FROM Customer ORDER BY CustID
DECLARE c2 CURSOR FOR SELECT ContFName, ContLName +
   FROM Contact WHERE CustID = .vCustID
OPEN c1

-- Get the first row of Customer data
FETCH c1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2,+
   vAddress INDICATOR iv3, vCity INDICATOR iv4, vState INDICATOR iv5, +
   vZipCode INDICATOR iv6, vPhone INDICATOR iv7
WHILE SQLCODE <> 100 THEN

   -- Display the Customer data and open cursor c2 to
   -- retrieve the matching contact data
   SET VAR vCityStateZip = (.vCity + ',' & .vState & .vZipCode)
   WRITE .vCustID, .vCompany
   WRITE .vAddress
   WRITE .vCityStateZip
   OPEN c2
   FETCH c2 INTO vFName INDICATOR i1, vLName INDICATOR i2
   WHILE SQLCODE <> 100 THEN
      SET VAR vFullName = (.vFName & .vLName)
      WRITE .vFullName
      FETCH c2 INTO vFName INDICATOR i1, vLName INDICATOR i2
   ENDWHILE

   -- Close cursor c2 and get the next row of
   -- Customer data
   CLOSE c2
   FETCH c1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2,+
      vAddress INDICATOR iv3, vCity INDICATOR iv4, vState INDICATOR iv5, +
      vZipCode INDICATOR iv6, vPhone INDICATOR iv7
ENDWHILE
DROP CURSOR c1
DROP CURSOR c2
CLEAR VAR vCustID, vCompany, vAddress, vCity, vState, +
   vZipCode, vPhone, vCityStateZip, vFName, vLName, vFullName
RETURN
```

Another small change also improves performance - instead of using SET VAR commands within the WHILE loops to concatenate city, state and zipcode together, and first and last name together, the concatenation operation can be done in the DECLARE CURSOR command. The concatenation in the DECLARE CURSOR reduces the number of commands that are repeated for each row and moves the work to the DECLARE CURSOR command.

```
*(CustRep4.RMD add the concatenation to the DECLARE CURSOR)
SET VAR vCustID INTEGER = NULL
SET VAR vCompany TEXT = NULL
SET VAR vAddress TEXT = NULL
SET VAR vCity TEXT = NULL
SET VAR vState TEXT = NULL
SET VAR vZipCode TEXT = NULL
SET VAR vPhone TEXT = NULL
SET VAR vCityStateZip TEXT = NULL
SET VAR vFName TEXT = NULL
SET VAR vLName TEXT = NULL
SET VAR vFullName TEXT = NULL

SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
IF vCheckCursor1 = 1 THEN
  DROP CURSOR c1
ENDIF
SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
IF vCheckCursor2 = 1 THEN
  DROP CURSOR c2
ENDIF

-- Replace SET VAR commands with expressions in
-- the DECLARE CURSOR
DECLARE c1 CURSOR FOR SELECT CustID, Company, +
   CustAddress, (CustCity + ',' & CustState & CustZip), +
   CustPhone FROM Customer ORDER BY CustID
DECLARE c2 CURSOR FOR SELECT (ContFName & ContLName) +
   FROM Contact WHERE CustID = .vCustID
OPEN c1

-- Retrieve and display the Customer data
FETCH c1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2, +
   vAddress INDICATOR iv3, vCityStateZip INDICATOR iv4, vPhone INDICATOR iv5
WHILE SQLCODE <> 100 THEN
   WRITE .vCustID, .vCompany
   WRITE .vAddress
   WRITE .vCityStateZip

   -- Retrieve and display the contact data
   OPEN c2
   FETCH c2 INTO vFullName INDICATOR i1
   WHILE SQLCODE <> 100 THEN
      WRITE .vFullName
      FETCH c2 INTO vFullName INDICATOR i1
   ENDWHILE

   -- Close cursor c2 and get the next row of
   -- Customer data
   CLOSE c2
   FETCH c1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2, +
      vAddress INDICATOR iv3, vCityStateZip INDICATOR iv4, vPhone INDICATOR iv5
ENDWHILE
DROP CURSOR c1
DROP CURSOR c2
CLEAR VAR vCustID, vCompany, vAddress, vCity, vState, +
   vZipCode, vPhone, vCityStateZip, vFName, vLName, vFullName
RETURN
```

The final change to improve performance is to use the RESET option on the OPEN c2 command instead of CLOSE c2. Overall, we have improved performance on this small set of rows by a full second. On a larger data set you can expect to see a greater performance improvement.

```
*(CUSTREP5.RMD)
--reset cursor 2 instead of close and open
SET VAR vCustID INTEGER = NULL
SET VAR vCompany TEXT = NULL
SET VAR vAddress TEXT = NULL
SET VAR vCity TEXT = NULL
SET VAR vState TEXT = NULL
SET VAR vZipCode TEXT = NULL
SET VAR vPhone TEXT = NULL
SET VAR vCityStateZip TEXT = NULL
SET VAR vFName TEXT = NULL
SET VAR vLName TEXT = NULL
SET VAR vFullName TEXT = NULL

SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
IF vCheckCursor1 = 1 THEN
  DROP CURSOR c1
ENDIF
SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
IF vCheckCursor2 = 1 THEN
  DROP CURSOR c2
ENDIF

SET VAR vCustID INTEGER
DECLARE c1 CURSOR FOR SELECT CustID, Company, +
   CustAddress, (CustCity + ',' & CustState & CustZip), +
   CustPhone FROM Customer ORDER BY CustID
DECLARE c2 CURSOR FOR SELECT (ContFName & ContLName) +
   FROM Contact WHERE CustID = .vCustID
OPEN c1
FETCH c1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2, +
   vAddress INDICATOR iv3, vCityStateZip INDICATOR iv4, vPhone INDICATOR iv5
WHILE SQLCODE <> 100 THEN
   WRITE .vCustID, .vCompany
   WRITE .vAddress
   WRITE .vCityStateZip

   -- Open cursor c2 with the RESET option,
   -- no CLOSE command is needed
   OPEN c2 RESET
   FETCH c2 INTO vFullName INDICATOR i1
   WHILE SQLCODE <> 100 THEN
      WRITE .vFullName
      FETCH c2 INTO vFullName INDICATOR i1
   ENDWHILE
   FETCH c1 INTO vCustID INDICATOR iv1, vCompany INDICATOR iv2, +
      vAddress INDICATOR iv3, vCityStateZip INDICATOR iv4, vPhone INDICATOR iv5
ENDWHILE
DROP CURSOR c1
DROP CURSOR c2
CLEAR VAR vCustID, vCompany, vAddress, vCity, vState, +
   vZipCode, vPhone, vCityStateZip, vFName, vLName, vFullName
RETURN
```

As you can see from the above examples, maximizing the work of the DECLARE CURSOR command provides significant performance improvements. The changes were small and they didn't involve a lot of time or programming effort, but these changes did result in definite performance benefits.

Customize the environment
In addition to optimizing your programming code, you can improve cursor performance by optimizing the environment. Obviously, code runs faster on a newer computer. Outside of

upgrading your hardware, however, certain R:BASE environment settings can be used to improve performance. These settings generally improve overall performance as well as cursor performance.

Look at the EXPLAIN.DAT output file generated by the MICRORIM_EXPLAIN variable to see the cursor query optimization. The OPEN command actually executes the query. Each query executed in your program puts an entry in EXPLAIN.DAT; for example, SELECT or UPDATE commands in the WHILE loop are reflected. You might also see a query reference to the SYS_RULES table, which is used for multi-user locking control.

By using EXPLAIN.DAT, you can easily see why using the RESET option on OPEN is faster. Normally, each OPEN redoes the query. When RESET is used, the query is only optimized once.

The EXPLAIN.DAT entries for the last two command files (CUSTREP4.RMD and CUSTREP5.RMD) from **Example 2** are shown here. The first entry shows nested cursors using the OPEN and CLOSE commands. The second entry shows using the RESET option on OPEN.

Cursor c1 on the Customer table is accessed sequentially, all rows in the table are retrieved, and no WHERE clause is used. If an indexed WHERE clause was used, EXPLAIN.DAT would show the index used. The second cursor on the Contact table does use an indexed WHERE clause to define the query. This query is redone each time the cursor is opened with a different vCustID value.

```
SortStrategy = DB_TAG  (internal=1)
SelectCost=1.  (OptimizationTime=0ms)
  Customer Sequential
SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286
SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286
SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286
....

SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286
SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286
SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286
SelectCost=1.  (OptimizationTime=0ms)
  SYS_RULES Sequential
```

The following EXPLAIN.DAT entry uses OPEN c2 RESET. The same query is used each time cursor c2 is accessed. The query does not need to be reoptimized each time the cursor is opened.

```
SortStrategy = DB_TAG  (internal=1)
SelectCost=1.  (OptimizationTime=0ms)
  Customer Sequential
SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286
SelectCost=1.  (OptimizationTime=0ms)
  SYS_RULES Sequential
```

For additional information on using the MICRORIM_EXPLAIN variable to see the cursor query optimization, refer to the "Environment Optimization" chapter within the Reference Index of the R:BASE Help.

8 Questions & Answers

Q. When should I use a cursor?
A. Use a cursor when it seems like the best way to get a task done. There are no rules or standards to say when you should use a cursor and when you shouldn't. Often the logic behind a cursor is easier to understand than the logic behind a complex SELECT or UPDATE command that works across a group of rows. Many programmers have replaced DECLARE CURSOR routines with a single INSERT, UPDATE or DELETE command, most often for performance reasons, but not all cursors can be replaced with a single SQL command.

Deciding to use a cursor will depend on your level of programming expertise and understanding of the task to be accomplished. First get the program to work; once it works, look at ways to make the program run more efficiently and faster.

Q. How do I make a cursor faster?
A. Using a DECLARE CURSOR is slower than using just a single SQL command working across a group of rows, but some tasks just can't be done without using a cursor. You can use certain techniques to maximize the performance of DECLARE CURSOR routines. However, just like deciding when to use a cursor, there are no rules or standards about improving the performance of a cursor.

One of the best ways to make a cursor faster is to move as much of the work as possible into the DECLARE CURSOR command itself. Let the cursor select as many columns as possible. If you are doing calculations for each row, see if you can use one of the SELECT functions with the GROUP BY option.

For additional suggestions to improve cursor performance, see the Optimizing Cursors chapter in this document.

Q. Should I use WHERE CURRENT OF or an explicit WHERE clause?
A. In terms of performance, there is very little difference between the two options. Not all cursors can be used with the WHERE CURRENT OF syntax. Getting the most out of your DECLARE CURSOR statement is more important in terms of performance than making your cursor an updateable cursor.

Q. I'm trying to UPDATE data using WHERE CURRENT OF and I get a syntax error. I have checked and double checked the syntax, and it is fine.
A. You get this error when you have a non-updateable cursor. A non-updateable cursor does not support use of WHERE CURRENT OF. Use an explicit WHERE clause to update the table instead of WHERE CURRENT OF.

Q. What is a non-updateable cursor?
A. A cursor knows what data to retrieve based on the SELECT statement that is part of the DECLARE CURSOR command. Like a regular SELECT command, the SELECT that is part of the DECLARE CURSOR can retrieve data from multiple tables or use a GROUP BY. It has all the features of the regular SELECT. However, only a single table SELECT with no GROUP BY is updateable; this option is the only one that guarantees the cursor is pointing to a single row in a table. If the cursor can't point back to and identify a single row, it doesn't know what to update.

Q. Is it faster to retrieve data inside my WHILE loop using the SET VAR command or the SELECT...INTO command?
A. It's just a little bit faster to retrieve additional data using a SET VAR command instead of the SELECT...INTO command. The SELECT has more overhead. The fastest way to retrieve column data into variables, however, is to retrieve whatever columns possible through the DECLARE CURSOR command. That method can be almost twice as fast as using either SET VAR or SELECT...INTO.

Q. My WHILE loop never ends. It just keeps repeating the last row.

A. FETCH, which sets SQLCODE, should be the last command in the WHILE loop. When no more data is available, SQLCODE is set to 100. If FETCH is the last command in the WHILE loop, the next command executed is the WHILE statement, which tests the current value of SQLCODE. Other SQL commands placed after the FETCH and before the ENDWHILE might reset SQLCODE to a value other than 100.

Also, if your WHILE condition is not SQLCODE <> 100, make sure you are checking the condition correctly. If the WHILE loop doesn't exit, the WHILE condition is never false. Use TRACE and set up watch variables to see what is happening with your variable values.

Q. Why won't WHENEVER work with DECLARE CURSOR?
A. WHENEVER is an SQL error trap command that executes a GOTO whenever the data not found situation (SQLCODE = 100) occurs. At first glance, WHENEVER seems ideal for use with a DECLARE CURSOR. However, if your DECLARE CURSOR routine uses any other SQL commands that can return a "data not found" error, such as SELECT, INSERT or UPDATE, the WHENEVER immediately exits the DECLARE CURSOR WHILE loop even though all the data has not been processed. The R:BASE error "No rows exist or satisfy the WHERE clause" is a "data not found" error and sets SQLCODE to 100.

Q. I use DECLARE CURSOR to find out if a row exists in a table. Is there a way to do this check faster?
A. If you only want to see if a row exists in a table, don't use DECLARE CURSOR. The DECLARE CURSOR command by itself doesn't check this. You need to OPEN the cursor and FETCH before you know if a row has been found. Instead use the SELECT command; SELECT INTO a variable and test the variable value, or test SQLCODE immediately after the SELECT command. If no row is found, SQLCODE is set to 100. Using just the SELECT command is much faster than using the DECLARE CURSOR.

Q. My DECLARE CURSOR command is giving me a syntax error. Is there an easy way to check the syntax?
A. First make sure the cursor name is in the correct place in the command. A common error is to use DECLARE CURSOR c1 instead of DECLARE c1 CURSOR. The SELECT part of the DECLARE CURSOR command can get quite complex, particularly when more than one table is involved. Test the SELECT part of the DECLARE CURSOR command at the R> Prompt, which executes just like a regular SELECT command. You can test and debug the SELECT part of your DECLARE CURSOR before putting it into the DECLARE CURSOR structure.

9 Useful Resources

. R:BASE Home Page:	http://www.rbase.com
. R:BASE X Home Page:	http://www.rbasex.com
. Up-to-Date R:BASE Updates:	http://www.rupdates.com
. Sample Applications:	http://www.rbasecommunity.com
. General R:BASE Syntax:	http://www.rsyntax.com
. Technical Documents - From The Edge:	http://www.razzak.com/fte
. More Sample Applications:	http://www.razzak.com/sampleapplications
. Education and Training:	http://www.rbaseuniversity.com
. Upcoming Events:	http://www.rbase.com/events
. R:BASE Beginners Tutorial:	http://www.rtutorial.com

Index

Notes