



Using Cursors to Reference Table Data

By proceeding with this courseware you agree with [these terms and conditions](#). Interskill Learning Pty. Ltd. © 2019



Objectives

Using Cursors to Reference Table Data

In this module you will see how cursors can be used to identify and retrieve one or more rows of data from a result table.

After completing this module you will be able to:

- Identify the Types of Cursors Available and State Their Purpose
- Describe the Steps Required When Using Cursors
- Determine the Attributes of a Cursor



In the modules so far, you have seen how the `SELECT` statement can be used to create a result table based on the criteria specified. With your application program you may want to only process a subset of these records, which is possible using the cursor to locate the relevant record to begin with.

In this course you will look at the types of cursor that can be used and how they are invoked.

```
EXEC SQL
DECLARE MYCURS CURSOR FOR
SELECT EMP_ID, LASTNAME, DEPT_ID
FROM EMPLOYEE
WHERE DEPT_ID LIKE = 'TST%'
FOR FETCH ONLY
END-EXEC.
```

This defines the cursor and the result table it is linked with.

```
EXEC SQL
OPEN MYCURS
END-EXEC.
```

The cursor then needs to be opened, which will build the result set.

```
EXEC SQL
FETCH MYCURS
INTO :EMP-ID, :L_NAME, :DEPT_ID
END-EXEC.
```

In this example the cursor is opened using the FETCH statement and the content of a row will be retrieved.

There are two types of cursors, a rowset-positioned cursor and a row-positioned cursor.

A rowset-positioned cursor retrieves zero, one, or more rows at a time, as a rowset, from the result table into host variable arrays. This type of cursor is discussed later in this module.

For now we will focus on the row positioned cursor. This type of cursor is a pointer to a particular row in a result set. It points to the next row to be passed to the application program. All that the application program has to do to return a row is ask for the row and nominate some host variables to put the data into.



```
EXEC SQL
  DECLARE EMP_CURSOR CURSOR FOR
  SELECT EMP_ID , NAME
  FROM EMPLOYEE
  WHERE NAME LIKE :SEARCH-NAME
END-EXEC.
. . .
MOVE SEARCH-STRING TO SEARCH-NAME.

EXEC SQL
  OPEN EMP_CURSOR
END-EXEC.
```

Here is another example of the code used when referencing a cursor. In a COBOL program, this code would reside in the data or procedure division.

The process with using a row positioned cursor is as follows:

- Declare the cursor
- Open the cursor; this causes Db2 to build the result set
- Fetch each row one at a time and then do something with it
- Close the cursor

You will look at each of these statements in turn as they are coded in a COBOL statement.





```
EXEC SQL
  DECLARE EMP_CURSOR CURSOR FOR
  SELECT EMP_ID , NAME
  FROM EMPLOYEE
  WHERE NAME LIKE :SEARCH-NAME
END-EXEC.
. . .
MOVE SEARCH-STRING TO SEARCH-NAME.

EXEC SQL
  OPEN EMP_CURSOR
END-EXEC.
```

The DECLARE CURSOR statement defines the name of the cursor and the SELECT statement associated with it.

With this statement you are telling Db2 that each row in the result set will be referenced by EMP_CURSOR.

Declaring the cursor does not cause the data to be selected from the table.



```
EXEC SQL
  DECLARE EMP_CURSOR CURSOR FOR
  SELECT EMP_ID , NAME
  FROM EMPLOYEE
  WHERE NAME LIKE :SEARCH-NAME
END-EXEC.
: : :
MOVE SEARCH-STRING TO SEARCH-NAME.
EXEC SQL
  OPEN EMP_CURSOR
END-EXEC.
```

The OPEN statement opens the cursor in readiness to process the rows from its result table. It is at this time that the result table is actually created, using the current values of any host variables that are specified in the SELECT statement.

In this example, a COBOL MOVE statement is being used to define a value (SEARCH-STRING) to the host variable SEARCH-NAME that appears in the DECLARE cursor SELECT statement area.



```
MOVE SEARCH-STRING TO SEARCH-NAME.  
  
EXEC SQL  
  OPEN EMP_CURSOR  
END-EXEC.  
  
EXEC SQL  
  FETCH EMP_CURSOR  
  INTO :EMP-ID, :NAME  
END-EXEC.
```

With the cursor now opened, we need to instruct the SQL on the information to be retrieved. This is achieved using the FETCH statement.

The FETCH statement positions the cursor on the first row of the result table and retrieves the information into the host variables specified, in this case EMP-ID and NAME.



```
EXEC SQL
  OPEN EMP_CURSOR
END-EXEC.

EXEC SQL
  FETCH EMP_CURSOR
  INTO :EMP-ID, :NAME
END-EXEC.

PERFORM UNTIL SQLCODE NOT= 0
  do something with the data
  .
  .
  .
  EXEC SQL
  FETCH EMP_CURSOR
  INTO :EMP-ID , :NAME
  END-EXEC.
END-PERFORM.

IF SQLCODE NOT = 100
  something went wrong!
END-IF.
```

Usually, FETCH statements are performed in a loop until all rows are processed or a certain number of rows are processed.

The code shown here is an example on how this can be achieved.

Note the following:

- FETCH is like a READ
- SQLCODE 100 is like END OF FILE





```
EXEC SQL
  DECLARE EMP_CURSOR CURSOR FOR
  SELECT EMP_ID, NAME
  FROM EMPLOYEE
  WHERE NAME LIKE :SEARCH-NAME
END-EXEC.

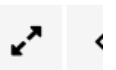
      .
      .
      .
MOVE SEARCH-STRING TO SEARCH-NAME.

EXEC SQL
  OPEN EMP_CURSOR
END-EXEC.

      .
      .
      .
EXEC SQL
  CLOSE EMP_CURSOR
END-EXEC.
```

When you have completed processing the data, you will need to enter a CLOSE statement to close the cursor. When this occurs the temporary result table will be destroyed. The only way to access the data after the cursor is closed is to open it again.

Note that closing cursors as soon as possible can improve performance.





```
EXEC SQL
  DECLARE EMP_CURSOR CURSOR FOR
  SELECT EMP_ID, NAME
  FROM EMPLOYEE
  WHERE NAME LIKE :SEARCH-NAME
END-EXEC.

      . . .
      . . .
MOVE SEARCH-STRING TO SEARCH-NAME.

EXEC SQL
  OPEN EMP_CURSOR
END-EXEC.

      . . .
EXEC SQL
  CLOSE EMP_CURSOR
END-EXEC.
```

Any open cursors will be automatically closed when the program ends. Other events that can result in the cursor being closed are:

- an abnormal termination of the batch job or online task
- SQL commit (although a WITH HOLD parameter can be placed on the DECLARE CURSOR statement to prevent this)
- Rollback
- CICS and IMS SYNCPOINT
- SQLCODE -911 or -913 (deadlock or timeout)



```
EXEC SQL
  DECLARE EMP_CURSOR CURSOR FOR
  SELECT NAME, SALARY
  FROM EMPLOYEE
  WHERE NAME LIKE :SEARCH-NAME
  FOR UPDATE OF NAME
END-EXEC.
..

EXEC SQL
  UPDATE EMPLOYEE
  SET NAME = :NAME
  WHERE CURRENT OF EMP_CURSOR
END-EXEC.
..

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

You can also use the cursor to indicate rows where an update will take place, or a row deletion will occur.

In the example shown here, the FOR UPDATE in the DECLARE CURSOR statement indicates that the NAME column should be updated. After the program has executed a FETCH statement to retrieve the current row, the UPDATE statement shown here can be used to modify that row's NAME data.

In the example at the bottom of the page, the DELETE statement will delete the row on which the cursor is currently positioned.

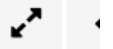


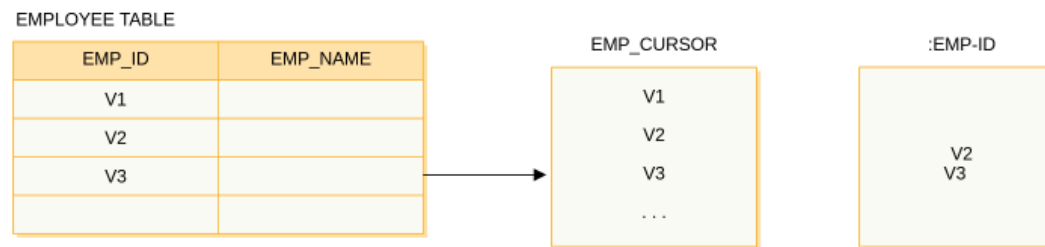
```
EXEC SQL
DECLARE EMP_CURSOR CURSOR FOR
SELECT EMP_ID,SALARY
FROM EMPLOYEE WHERE DEPT_ID = :DEPT-ID
FOR UPDATE OF SALARY
END-EXEC.

MOVE TPO-DEPT-ID TO DEPT-ID.
EXEC SQL
OPEN EMP_CURSOR
END-EXEC.
EXEC SQL
FETCH EMP_CURSOR INTO :EMP-ID, :SALARY
END-EXEC.
PERFORM UNTIL SQLCODE NOT = 0
EXEC SQL
UPDATE EMPLOYEE
SET SALARY=SALARY*1.1 WHERE CURRENT OF EMP_CURSOR
END-EXEC.
write the new and old salary to a file
EXEC SQL
FETCH EMP_CURSOR INTO :EMP-ID,:SALARY
END-EXEC.
```

Here is a program sample that reads through a table and updates all the salaries in a department (:DEPT-ID) with a 10% raise. Look through the code and notice that all four steps in cursor processing are here:

- DECLARE cursor
- OPEN the cursor
- FETCH information
- CLOSE the cursor when finished





For many rows:

```
EXEC SQL  
  DECLARE EMP_CURSOR CURSOR FOR  SELECT EMP_ID  
    INTO :EMP-ID  
    FROM EMPLOYER  
  END EXEC  
  ...  
EXEC SQL  
  OPEN EMP_CURSOR  
  END EXEC
```

From Version 7 of DB2 onwards, cursors can be scrolled forwards or backwards.

From Version 7 of Db2 onwards, cursors can be scrolled forwards or backwards. With previous versions of Db2, cursors could only be scrolled or processed sequentially forwards.

You will now look at declaring and opening a scrollable cursor, fetching and moving within the cursor, and the difference between sensitive and insensitive cursors.

Click Play to see how cursor use has changed.



```
DECLARE C1 SENSITIVE SCROLL CURSOR
FOR SELECT STAFF_ID, NAME, LOCATION
FROM EMPLOYEE
...
OPEN C1
...
```

The temporary result table is created in the work file database, and will be dropped when the cursor is closed.

EMPLOYEE TABLE

STAFF_ID	NAME	POSCODE	POSDECS	LOCATION
100	BLACK	A1	DBA	SYDNEY
400	SMITH	B2	OPERATOR	NEW YORK
300	JONES	A1	SYS ADMIN	LONDON

TEMPORARY TABLE

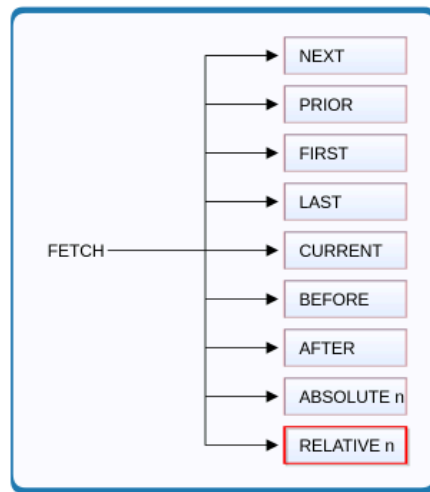
STAFF_ID	NAME	LOCATION
100	BLACK	SYDNEY
400	SMITH	NEW YORK
300	JONES	LONDON

To open a cursor that will use scrolling, the keyword SCROLL is used on the DECLARE CURSOR statement.

When the DECLARE CURSOR...SCROLL and OPEN CURSOR are executed, unless the cursor is against a single base table, a temporary table in the work file database is created to hold the result set.

Click Play to see how the keyword SCROLL is used in the DECLARE CURSOR statement.

FETCH cursor syntax



Will fetch the row that is n rows away from the last row fetched.

```
EXEC SQL
FETCH ABSOLUTE +3 MYCURS INTO :HV_NAME
END-EXEC

EXEC SQL
FETCH RELATIVE 2 MYCURS INTO :HV_SALARY
END-EXEC

EXEC SQL
FETCH AFTER FROM MYCURS
END-EXEC
```

With the cursor having been declared as a scrollable cursor, the FETCH statement can now be used to define where the cursor is to be positioned in the result table, before it begins processing.

The information shown here represents the SQL parameters that allow the cursor to be repositioned for a row-positioned cursor.

Mouse-over the syntax for a description of that parameter.


```
DECLARE C1 INSENSITIVE SCROLL CURSOR
FOR SELECT STAFF_ID, NAME, LOCATION
FROM EMPLOYEE
...
OPEN C1
...
```

Data in the EMPLOYEE table is deleted in the second row, but the cursor is unaware of the update as, due to the INSENSITIVE flag, the temporary table is not updated accordingly.

EMPLOYEE TABLE

STAFF_ID	NAME	POSCODE	POSDECS	LOCATION
100	BLACK	A1	DBA	SYDNEY
300	JONES	A1	SYS ADMIN	LONDON






TEMPORARY TABLE

STAFF_ID	NAME	LOCATION
100	BLACK	SYDNEY
400	SMITH	NEW YORK
300	JONES	LONDON

When using scrollable cursors you must assign the cursor a sensitivity level. This describes whether the result table is updateable or read-only. Until now, we have left our examples blank when referring to scrollable cursors, which indicates that they have been using a default value of ASENSITIVE. This is a flexible sensitivity level that will perform as an update, or read-only cursor depending on the environment it is dealing with.

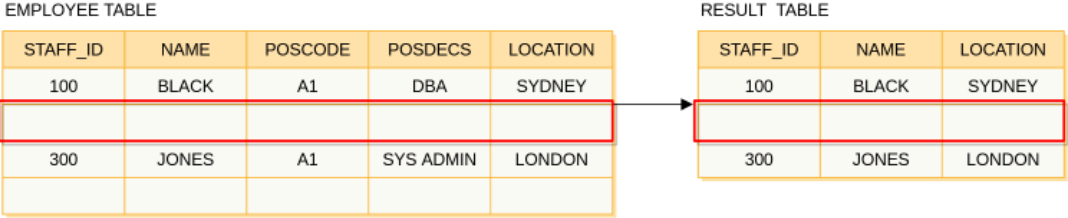
An example of using an INSENSITIVE cursor is shown here and indicates that it is read only. This means that the cursor result table will not display any changes such as inserts, updates or deletes made to the base data.

Click Play to see how the INSENSITIVE attribute works.

```
DECLARE C1 SENSITIVE DYNAMIC SCROLL CURSOR
FOR SELECT STAFF_ID, NAME, LOCATION
FROM EMPLOYEE
...
OPEN C1
...
```

Data in the EMPLOYEE table is deleted in row 2 and the cursor is aware of the update as the result table is updated accordingly.



The SENSITIVE attribute is used when you want the cursor result table to be aware of any changes made to the base table (such as an update or delete). To further fine tune this feature, a STATIC or DYNAMIC parameter is usually coded alongside it (although if not supplied, then DYNAMIC is the default).

DYNAMIC specifies that following the opening of the cursor, the size of the result table might change as a result of inserts or deletes being performed on the base table. These changes may be through the same application process associated with the cursor, or as a result of a commit process from another application. STATIC is used when you do not want the cursor result table modified to reflect actions occurring in the base table. This means that an INSERT in the base table will not be reflected in the cursor result table, and an UPDATE or DELETE will display as a hole in the cursor result table.

```
DECLARE MYCURS SENSITIVE STATIC SCROLL  
FETCH INSENSITIVE
```

- The cursor is updateable
- Recognizes updates or deletes within cursor
- Updates and deletes made to the base table outside of the cursor are not displayed
- Any inserts are not recognized

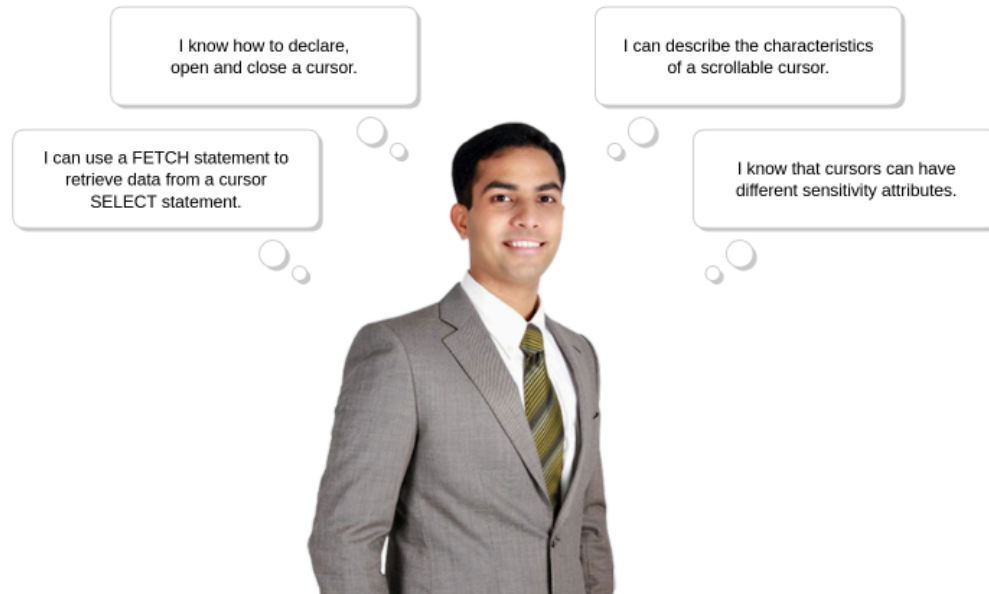
```
DECLARE MYCURS SENSITIVE STATIC SCROLL  
FETCH SENSITIVE
```

- The cursor is updateable
- Recognizes updates or deletes within cursor
- Committed updates and deletes inside and outside of the cursor are recognized
- Any inserts are not recognized

```
DECLARE MYCURS INSENSITIVE SCROLL  
FETCH INSENSITIVE
```

- Cursor is read-only
- Cursor result table does not recognize updates or deletes in base table

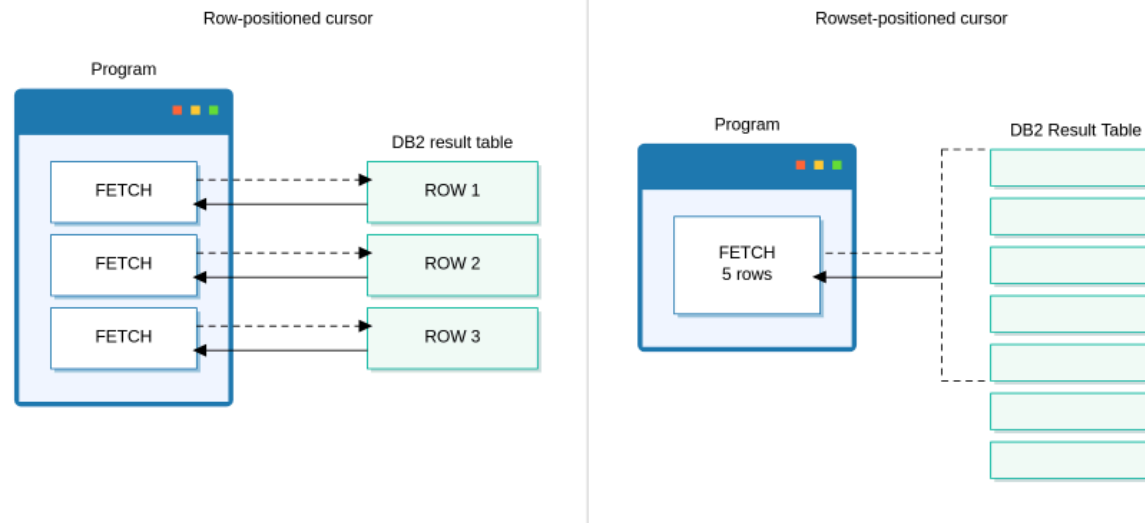
A number of DECLARE CURSOR and FETCH sensitivity combinations can be used, depending on your requirements. The table here provides details on how data is managed with these commonly used combinations.



A scrollable cursor is closed in the same way as a non-scrollable cursor and any temporary table is cleaned.

To summarize: a scrollable cursor gives more flexibility at the cost of complexity; you fetch into host variables and can update and delete as with a non-scrollable cursor, but you can also revisit or skip cursor rows.

Whether you see the effect of earlier or other updates in those rows will depend on your use of the `ASENSITIVE`, `SENSITIVE`, and `INSENSITIVE` keywords.



The row-positioned cursor discussed so far performs a fetch on one row at a time and then may, or may not be repositioned for processing other rows. Depending on the result you are aiming to achieve, a multi-row fetch may be more appropriate using a ROWSET positioned cursor.

In this section you will see the similarities and differences between row-positioned and rowset-positioned cursor.



```
DECLARE MYCURS CURSOR
WITH ROWSET POSITIONING FOR
SELECT EMP_ID, L_NAME
FROM EMPLOYEE
. . . . .

OPEN MYCURS
. . . . .

FETCH ROWSET
STARTING AT ABSOLUTE -2 FROM MYCURS
FOR 5 ROWS
INTO :EMP :LNAME;
. . . . .

CLOSE MYCURS
```

The process involved with dealing with rowset-positioned cursors is the same as row-positioned cursors.

1. The rowset cursor needs to be declared
2. The rowset cursor needs to be opened
3. Multiple row fetch statements are invoked
4. The rowset cursor is closed



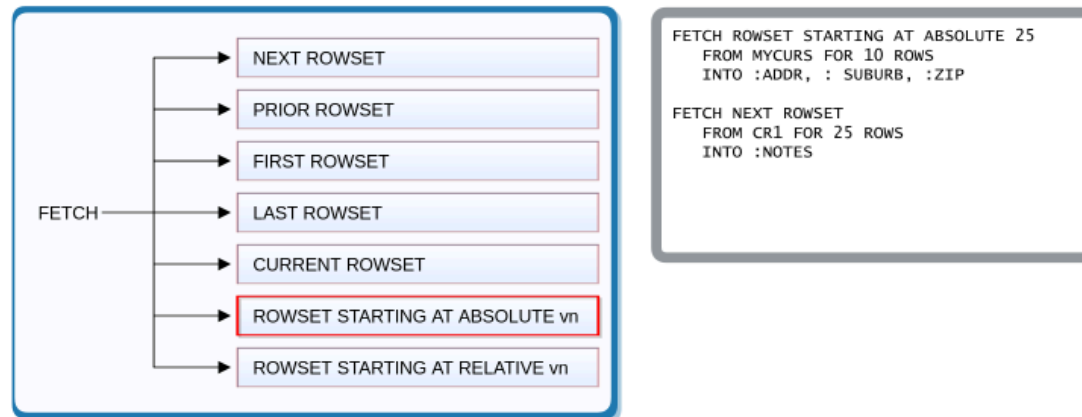


```
01 OUTPUT VARS.  
 05 L_NAME OCCURS 6 TIMES.  
    49 L_NAME_LGTH PIC S9(4) USAGE COMP.  
    49 L_NAME_DATA PIC X(50).  
 05 EMP-ID PIC S9(9) COMP OCCURS 6 TIMES.  
  
PROCEDURE DIVISION.  
EXEC SQL  
  DECLARE MYCURS CURSOR WITH ROWSET POSITIONING FOR  
  SELECT L_NAME, EMP_ID  
  FROM EMPLOYEE  
END-EXEC.  
EXEC SQL  
  OPEN MYCURS  
END-EXEC.  
EXEC SQL  
  FETCH NEXT ROWSET FROM MYCURS FOR 6 ROWS  
  INTO :L_NAME, :EMP_ID  
END-EXEC.
```

Using a rowset cursor, the associated FETCH statement copies the ROWSET column values into one of the following:

- Host variable arrays that are declared in your program as shown in the COBOL example above, or into
- Dynamically-allocated arrays whose storage addresses are put into an SQL descriptor area (SQLDA)





Will fetch the rowset whose first row is the row number indicated by either a host variable (v) or an integer (n).

Positioning of the rowset cursor requires similar parameters to those discussed previously.

Mouse-over the code for a description of its purpose.





Updating an individual line within a ROWSET

```
UPDATE EMPLOYEE  
SET BONUS = YES  
FOR CURSOR MYCURS  
FOR ROW 10 OF ROWSET
```

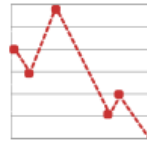
Updating all lines within a ROWSET

```
UPDATE CUST  
SET STATUS = VIP  
WHERE CURRENT OF MYCURS
```

Just because you may have fetched multiple rows of data using rowset doesn't mean that you can't update specific rows.

In the top example, an UPDATE statement references a specific row within the ROWSET to update. The lower example will update all rows in the ROWSET.





Reduced CPU usage



Dollars saved



Greater application efficiency

You should evaluate your existing code to determine whether multi-row fetch using ROWSET is an option as it provides the following benefits:

- The number of operations required to fetch multiple rows of data is reduced using ROWSET, which greatly reduces CPU usage
- The performance in your application program will vary depending on the SELECT and FETCH operations and the amount of data being accessed, but generally it will be improved using ROWSET

SQLCA

SQLWARN1	Indicates a scrollable or non-scrollable cursor.
SQLWARN4	Displays details relating to the sensitivity of the cursor. For example, whether it is insensitive (I), sensitive static (S), or sensitive dynamic (D).
SQLWARN5	Indicates the functionality of the cursor; that is whether it is read-only, or can be used for deleting and updating.
SQLERRD(1) SQLERRD(2)	These two fields contain details about the number of rows in the result table of a cursor when the cursor is positioned after the last row (when SQLCODE = 100). These codes will not be set for dynamic scrollable cursors.
SQLERRD(3)	If the SELECT statement of the cursor contains a data change statement, then this field will display the number of rows in the result table.



With so many attributes and settings that can be defined for a cursor, there may be situations where you need to identify exactly what these specifications are.

After you open a cursor, you can check the SQLWARN and SQLERRD fields of the SQL Communications Area (SQLCA) by invoking the assembler subroutine DSNTIAR from your program, or by including the GET DIAGNOSTICS SQL statement in your code.

The table displayed here indicates the SQLCA fields containing some cursor-related data.