IBM DATABASE 2 (DB2) for z/OS
Version 8
Multi-Row Insert and Fetch
Line Item 382
Programming Functional Specification (PFS)

LI382 PFS

Chris Crone

Department D3T, 090/B209
DB2 Development
Santa Teresa Laboratory

March 11, 2002

IBM DATABASE 2 (DB2) for z/OS Version 8 Multi-Row Insert and Fetch Line Item 382 Programming Functional Specification (PFS)

Document Control Information

Document Owner:	Chris Crone Line Item Owner
Document Owner Node(ID):	Chris Crone/Santa Teresa/IBM
Document Owning Department:	D3T
Document Source Location:	LI382.ZIP in CMVC
Document Object Location:	None.
Document Reviewers:	

Name	Dept, Role <(Optional) reviewer>	Node(ID)
Curt Cotner	J34, DB2 Development, Optional	Curt Cotner/Santa Teresa/IBM
Jay Yothers	KIH, DB2 Development, Optional	Jay Yothers/Santa Teresa/IBM
Claire McFeely	OF2, DB2 Development, Required	Claire McFeely/Santa Teresa/IBM
Marion Farber	OF2, DB2 Development, Required	Marion Farber/Santa Teresa/IBM
Janet Chen	FEY, DB2 Quality Assurance , Required	Janet Chen/Santa Teresa/IBM
Karelle Cornwell	M55, DB2 Development, Required	Karelle Cornwell/Santa Teresa/IBM
Ester Mote	M55, DB2 Development, Required	Ester Mote/Silicon Valley/IBM
Casey Young	M60, DB2 Performance, Required	Casey Young/Santa Teresa/IBM
Judy Tobias	J64, DB2 Information Development, Required	Judy Tobias/Santa Teresa/IBM
Ching-Sen Tseng	OB9, DB2 Quality Assurance, Required	Ching-Sen Tseng/Santa Teresa/IBM
Margaret Dong	L09, DB2 Development, Required	Margaret Dong/Santa Teresa/IBM
Maria Sueli Almeida	M05, DB2 System Test, Required	Maria Sueli Almeida/Silicon Valley/IBM
Meg Bernal	D3T, DB2 Development, Required	Meg Bernal/Santa Teresa/IBM
Yumi Tsuji	6SW, DB2 Development, Required	Yumi Tsuji/Santa Teresa/IBM
Daya Vivek	D3T, DB2 Development, Required	Daya Vivek/Silicon Valley/IBM
Hsiuying Cheng	6S7, DB2 Development, Required	Hsiuying Cheng/Santa Teresa/IBM
Wendy Koontz	L09, DB2 Development, Required	Wendy Koontz/Boulder/IBM
Tammie Dang	D3T, DB2 Development, Required	Tammie Dang/Santa Teresa/IBM

Document Approvers: *document owner and document owning manager*

Name	Dept, Role	Node(ID)
Chris Crone	D3T, Line Item Owner	Chris Crone/Santa Teresa/IBM
Peter Wu	D3T, Development Manager	Peter Wu/Santa Teresa/IBM

Document Updaters: *delegated authority by the owner to update the document*

Name	Dept, Role	Node(ID)
Line Item Team Members		

Document Distribution: Availability of approved documents is announced on DB2NEWS.

Document Printing

All files are in CMVC. The document may be printed using:

NOTE:

The hard copy version of this document is FOR REFERENCE ONLY.

It is the responsibility of the user to ensure that they have the current version. Any outdated hard copy is invalid and must be removed from possible use. It is also the responsibility of the user to ensure the completeness of this document prior to use.

Change History

Contents

Tables

Figures

Chapter 1. Description

Abstract

This line item deals with implementing multiple-row processing for both the FETCH and the INSERT statement in DB2 for z/OS. Prior to Version 8, a user would have to execute a separate SQL FETCH statement for each row of data that the application required from the database. Likewise if an application needed to insert several rows, that application would have to execute a separate SQL INSERT statement for each row being stored into the database. For local processing, the execution cost was the multiple trips between the application and the database engine. For distributed, the performance cost would be the multiple trips into the database engine plus the network cost to send each request. In some cases, block fetching introduced in V2.2 and V2.3 mitigated the network costs for the FETCH statement.

Functional Description

DB2 for z/OS V8 will support multi-row insert and fetch statements. These statements will be described in detail later in this document. To introduce these statements we will discuss some examples in this section:

FETCH

Fetch with a host-variable-array:

```
FETCH FIRST ROWSET STARTING AT ABSOLUTE 10 FROM CURS1
FOR 6 ROWS
INTO :hva1, :hva2;
```

The multiple-row FETCH is implemented as a static SQL statement.

With this line item, a single FETCH statement can be used to retrieve multiple rows of data from the result table of a query as a rowset. A rowset is a group of rows that are grouped together and operated on as set. For example, you may fetch the next rowset, or update the current rowset. The program or application controls how many rows are returned on a single FETCH statement. Fetching multiple rows of data can be done with both serial and scrollable cursors. New syntax on the FETCH statement allows specification of the number of rows to be returned in the rowset.

FOR 6 ROWS

This clause specifies that with a single SQL statement in the application program, DB2 will fetch the stated "6" rows.

INTO :hva1, :hva2;

The first host-variable-array corresponds to the first column's output. The second host-variable-array corresponds to the second column's output, and so on. A host-variable-array is an array in which each element of the array corresponds to a value for a column.

INSERT

In releases prior to DB2 for z/OS V8, SQL INSERT statement inserts one row of data into a table or view. With this line item, DB2 for z/OS is implementing an enhanced multi-row INSERT statement that will insert one or more rows into a table or view with one SQL statement.

There are 2 forms of multiple-row INSERT: one static, and one dynamic form. Here are some examples:

1. Static INSERT with host-variable-arrays:

   ```
   INSERT INTO T FOR :n ROWS
   VALUES(:hva1, :hva2);
   ```

2. Dynamic INSERT with host-variable-arrays:

   ```
   stmt = 'INSERT INTO T VALUES(?, ?)';
   attrvar = 'FOR MULTIPLE ROWS ATOMIC';
   PREPARE my_insert ATTRIBUTES :attrvar FROM :stmt;
   EXECUTE my_insert FOR :hv ROWS USING (:hva1, :hva2);
   ```

The multiple-row INSERT is implemented as either a static or dynamic SQL statement. Note that the FOR "n" ROWS clause is supplied on the EXECUTE statement for the dynamic form of INSERT instead of on the INSERT statement itself, as it is on the static form of INSERT. If "n" is greater than or equal to 1, then the parameter marker represents a host-variable-array.

FOR n ROWS

The maximum number of rows that can be inserted with a single INSERT statement is 32767. The input data for these multiple rows will be provided with new host-variable-arrays where each array represents the multiple rows of a single column. The VALUES or USING DESCRIPTOR clause will allow specification of multiple rows of data. For exampe, assuming :hva1 and :hva2 represent host variable arrays, the following VALUES clause may be used to specify the multiple values for an insert statement: **VALUES (:hva1, :hva2)**

ATOMIC or NOT ATOMIC

An option will be provided so that the application can specify if it wants the multiple-row INSERT to succeed or fail as a unit, or if it wants DB2 to proceed despite a partial (one or more rows) failure. The SQL clause to do this is **ATOMIC or NOT ATOMIC** where ATOMIC specifies that if the insert for any row fails, then all changes made to the database by any of the inserts, including changes made by successful inserts are undone. This is the default. When NOT ATOMIC is specified, the inserts are processed independently. This means that if one or more errors occurs during the execution of an INSERT statement, then processing continues and any changes made during the execution of the statement are not backed out.

Using host-variable-arrays

To use a multiple-row FETCH or INSERT statement with a host-variable-array per column, the application must define one or more host-variable-arrays that can be used by DB2. Each language has its own conventions and rules for defining a host-variable-array.

A host-variable-array corresponds to the values for one column of the result table for FETCH, or column of data to be inserted for INSERT. The first value in the array corresponds to the value for that column for the first row, the second value in the array corresponds to the value for the column in the second row, and so on. DB2 determines the attributes of the values in the array based on the declaration of the array.

Host-variable-arrays are used to return the values for a column of the result table on FETCH, or to provide values for a column on INSERT.

GET DIAGNOSTICS

The **GET DIAGNOSTICS** statement will be added to enable applications to retrieve diagnostics information about statements that have been executed.

This statement compliments and extends the diagnostics that are available in the SQLCA. The following example demonstrates the use of this new statement:

```
In an application, use GET DIAGNOSTICS to determine how many rows were
updated.

  long rcount;

  EXEC SQL UPDATE T1 SET C1 =C1 +1;

  EXEC SQL GET DIAGNOSTICS :rcount = ROW_COUNT;

After execution of this code segment, rcount will contain the number of
rows that were updated.
```

SQLCA

After a multiple-row INSERT or multiple-row FETCH statement, information is returned to the program through the SQLCA. The SQLCA will be set as follows:

- SQLCODE - SQLCODE of last error
- SQLSTATE - SQLSTATE of last error
- SQLERRD3 - actual number of rows inserted (in the case of INSERT), or the number of rows returned in the case of a multi-row FETCH
- SQLWARN - accumulation of flags set during any single insert

References to Host Variables

A value of -3 for an indicator variable indicates that values were not returned for the row because a hole was detected.

The purpose of an indicator variable is to indicate when the associated value is the null value, or that values were not returned because a hole was detected. The negative value is:

- -1 if the value selected was the null value
- -2 if the null value was returned due to a numeric conversion or arithmetic expression error that occurred in the SELECT list of an outer SELECT statement
- -3 if the null value was returned because a hole was detected for the row on a multiple row FETCH, and values were not returned for the row. In cases where -3 is set to indicate a null, SQLSTATE 02502, SQLCODE +222, will also be returned for that row. The value of -3 is only used for multiple-row FETCH statements, otherwise the only indication of the hole is the warning that is returned (SQLSTATE 02502, SQLCODE +222).

If indicator variables are not provided for a multiple-row FETCH statement, and a hole is detected, an error is returned (SQLSTATE 24519, SQLCODE -247).

Chapter 2. Usage Reference

SQL Statements

Language Elements

Add the following terms to the glossary:

- **cursor** A named control structure used by an application program to point to one or more specific rows within a set of rows of the result table. The cursor is used to retrieve rows from the result table (with a FETCH statement), and possibly to make updates or deletes to corresponding rows in the database. A cursor is defined with a DECLARE CURSOR statement. A cursor can be defined to always return a single row, or to possibly return multiple rows depending on what is specified on the FETCH statement.
- **host-variable-array** An array in which each element of the array corresponds to a value for a column. The dimension of the array determines the maximum number of rows that the array can be used for.
- **rowset** A set of rows that is retrieved through a multiple-row fetch.
- **rowset cursor** A cursor defined such that one or more rows can be returned for a single FETCH statement as a rowset, and the cursor is positioned on the set of rows fetched. With a rowset cursor when the number specified in the FOR n ROWS clause of FETCH is greater than 1 the cursor can be positioned on more than one row. Each row of the cursor position for a rowset cursor can be referenced in subsequent positioned DELETE and UPDATE statements. A FETCH statement for a rowset cursor specifies a rowset-positioned fetch orientation clause, and can indicate the desired number of rows for the rowset.

Host-Variable-Arrays in C/C++, COBOL, and PL/I

A host-variable-array is an array in which each element of the array contains a value for the same column . The first element in the array corresponds to the first value, the second element in the array corresponds to the second value, and so on.

A host-variable-array can only be referenced in the SQL FETCH statement when using a multiple row fetch or in an INSERT statement when using a multiple row insert. Host-variable-arrays are defined by statements of the host language, as explained in the Application Programming and SQL Guide.

The form of a host-variable-array reference is similar to the form of a host variable reference. The reference :COL1 :COL1IND is a host-variable-array reference if COL1 designates an array. If COL1 designates an array, COL1IND must be a one dimensional array of small integer host variables. The dimension of the host-variable-array must be less than or equal to the dimension of the indicator array (DSNH5011I). If an indicator array is not specified, no variable of the host-variable-array has an indicator variable. The following diagram specifies the syntax for references to a host-variable-array:

```
►►──:host-identifier──┬────────────────────────────────┬──────────────►◄
                      │ ┌─INDICATOR─┐                  │
                      └─┴───────────┴─:host-identifier─┘
```

In the following example, COL1 is the main host-variable-array and COL1IND is its indicator array. Assuming that COL1 has 10 elements (for fetching a single column of data for multiple rows of data), then COL1IND must also have 10 entries.

```
 EXEC  SQL FETCH C1 FOR 5 ROWS

       INTO :COL1 :COL1IND

       END-EXEC.
```

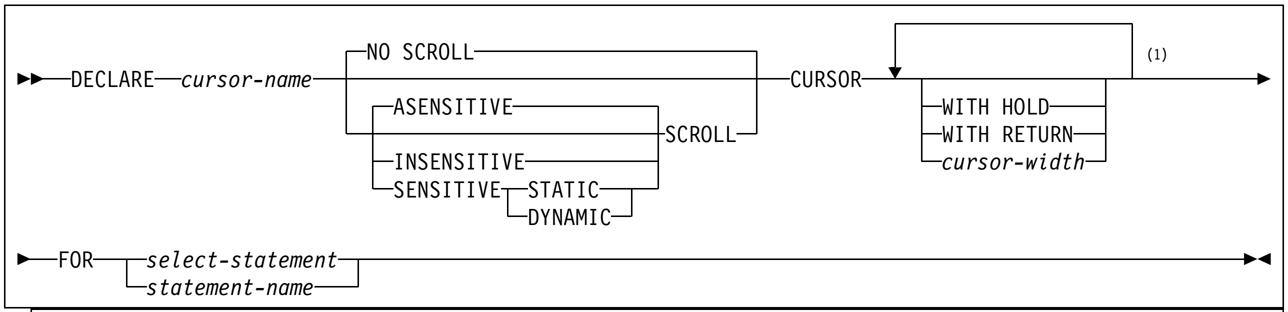
References to Host Variables

This line item introduces a new value of -3 for indicator variables to indicate that values were not returned for the row because a hole was detected.

The purposes of the indicator variable is to specify the null value. The negative value is:

- -1 if the value selected was the null value
- -2 if the null value was returned due to a numeric conversion or arithmetic expression error that occurred in the SELECT list of an outer SELECT statement.
- -3 if the null value was returned because a hole was detected for the row on a multiple row FETCH, and values were not returned for the row. The value of -3 is only used for multiple-row FETCH statements, otherwise the only indication of the hole is the warning that is returned (SQLSTATE 02502, SQLCODE +222).

DECLARE CURSOR Statement



Note:

1. The same clause must not be specified more than once.

Figure 1. DECLARE CURSOR Statement

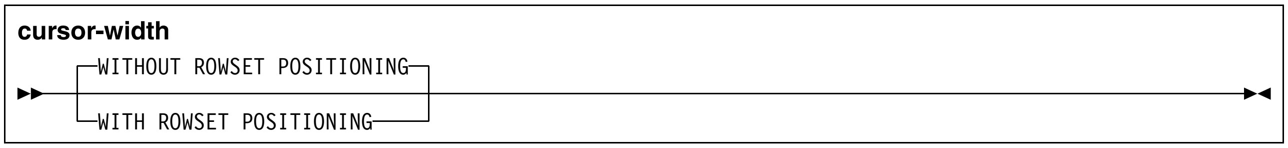


Figure 2. DECLARE CURSOR - cursor-width

SCROLL or NO SCROLL

Specifies whether the cursor is scrollable.

SCROLL
Specifies that the cursor is scrollable. For a scrollable cursor, whether the cursor has sensitivity to inserts, updates, or deletes depends on the cursor sensitivity option in effect for the cursor. If SCROLL is specified and neither ASENSITIVE nor SENSITIVE is specified, then the cursor is read-only and behaves as INSENSITIVE.

NO SCROLL
Specifies that the cursor is not scrollable. This is the default.

cursor-width

Specifies whether multiple rows of data can be accessed as a rowset on a single FETCH statement for this cursor. The default is WITHOUT ROWSET POSITIONING

WITHOUT ROWSET POSITIONING
Specifies that the cursor can only be used to return a single row for each FETCH statement, and that the FOR n ROWS clause cannot be specified on a FETCH statement for this cursor (SQLSTATE 24523, SQLCODE -249).[1]

WITH ROWSET POSITIONING
Specifies that this cursor can be used to return either a single row or multiple rows, as a rowset, with a single FETCH statement. Cursors declared WITH ROWSET POSITIONING may also be used with row positioned FETCH statements.[2]

Examples

```
Example 1:  Declare C1 as the cursor of a query to retrieve a rowset from the
table DEPT. The prepared statement is MYCURSOR.

EXEC SQL DECLARE C1 CURSOR WITH ROWSET POSITIONING FOR MYCURSOR;
```

OPEN CURSOR

The OPEN CURSOR statement is changed to return the following information for all cursors:

rowset accessibility
This information will be returned in DB2_SQL_ATTR_CURSOR_ROWSET which is available via the GET DIAGNOSTICS statement. The information will be returned as follows:

- Y = enabled for rowset fetching
- N = rowset fetching not supported

ALLOCATE CURSOR

The ALLOCATE CURSOR statement is changed to return the following information for all cursors:

rowset accessibility
This information will be returned in DB2_SQL_ATTR_CURSOR_ROWSET which is available via the GET DIAGNOSTICS statement. The information will be returned as follows:

1. Note: Single row access (WITHOUT ROWSET POSITIONING) refers to how data is fetched from the database engine. For remote access, data may be blocked and returned to the client in blocks. This blocking is subject to existing rules and restrictions.
2. Note: ROWSET POSITIONING refers to how data is fetched from the database engine. For remote access, if any row qualifies, at least 1 row will be returned as a rowset. The size of the rowset depends on the number of rows specified on the FETCH statement, and on the number of rows that qualify. Data may be blocked and returned to the client in blocks. This blocking is subject to existing rules and restrictions.

- Y = enabled for rowset fetching
- N = rowset fetching not supported

FETCH Statement

The FETCH Statement positions the cursor on a row of the result table. It can return zero, one, or multiple rows and assigns the values of the rows to host variables if there is a target specification.

There are two forms of this statement

- **single row fetch:** retrieves data from a single row of the result table.
- **multiple row fetch:** retrieves one or more rows from the result table.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared. Multiple row fetch is not supported in REXX, FORTRAN[3], or SQL Procedure applications due to the lack of support for arrays in these languages.

3. ASSEMBLER and other languages are supported, but this support is limited to statements that allow USING DESCRIPTOR. The precompiler will not recognize host-variable-arrays except in C/C++, COBOL, and PL/I.

FETCH

```
                                                    ┌─FROM─┐
►►──FETCH──┬─────────────┬──┤ fetch-orientation ├──┴──────┴──cursor-name──┬───────────────────────────┬──►◄
           ├─INSENSITIVE─┤                                                 ├─┤ single-row-fetch ├───┤
           │          (1)│                                                 └─┤ multiple-row-fetch ├─┘
           └─SENSITIVE───┘
```

fetch-orientation:

```
           (2)
├───┬─BEFORE─────────────────┬───────────────────────────────────────────────────────────────────────┤
    │       (2)              │
    ├─AFTER──────────────────┤
    ├─┤ row-positioned ├─────┤
    └─┤ rowset-positioned ├──┘
```

row-positioned:

```
    ┌─NEXT───────────────────────────┐
├───┼────────────────────────────────┼──────────────────────────────────────────────────────────────┤
    ├─PRIOR──────────────────────────┤
    ├─FIRST──────────────────────────┤
    ├─LAST───────────────────────────┤
    ├─CURRENT────────────────────────┤
    ├─ABSOLUTE──┬─host-variable──────┤
    │           └─integer-constant─┘ │
    └─RELATIVE──┬─host-variable──────┘
                └─integer-constant─┘
```

rowset-positioned:

```
├───┬─NEXT ROWSET────────────────────────────────────────┬──────────────────────────────────────────┤
    ├─PRIOR ROWSET───────────────────────────────────────┤
    ├─FIRST ROWSET───────────────────────────────────────┤
    ├─LAST ROWSET────────────────────────────────────────┤
    ├─CURRENT ROWSET─────────────────────────────────────┤
    └─ROWSET STARTING AT──┬─ABSOLUTE─┬──┬─host-variable──────┘
                          └─RELATIVE─┘  └─integer-constant─┘
```

Note:

1. The default depends on the sensitivity of the cursor. If INSENSITIVE or SENSITIVE is specified, either a single-row-fetch or multiple-row-fetch clause must be specified (SQLSTATE 42601, SQLCODE -104).

2. If BEFORE or AFTER is specified then SENSITIVE, or INSENSITIVE must not be specified (SQLSTATE 42601, SQLCODE -199).

FETCH

single-row-fetch:

```
          ┌─,──────────┐
├──┬─INTO─▼─host-variable─┴─────────────────────────────────────────────────────────────┤
   │                      (1)              │
   └─INTO DESCRIPTOR── descriptor-name──┘
```

multiple-row-fetch:

```
   (2)
├──────────────────────────────────────────┬─────────────────────────┬───────────────────────┤
         └─FOR──┬─host-variable────┬──ROWS─┘  │        ┌─,───────────────┐   │
                └─integer-constant─┘            │        ▼                 │   │
                                                ├─INTO────host-variable-array─┴───┤
                                                │                      (1)        │
                                                └─INTO DESCRIPTOR── descriptor-name─┘
```

Note:

1. "USING DESCRIPTOR" may be used as a synonym for "INTO DESCRIPTOR"

2. This clause is optional. If this clause is not specified, and either a rowset size has not been established yet, or a row positioned FETCH statement was the last type of FETCH statement issued for this cursor, the rowset size will implicitly be One. If the last FETCH statement issued for this cursor was a rowset positioned FETCH statement, and this clause is not specified, the rowset size will be the same size as the previous rowset positioned FETCH.

Figure 3. FETCH Statement

AFTER

Positions the cursor after the last row of the result table, values are not assigned to host variables. The number of rows of the result table is returned in the SQLERRD1 and SQLERRD2 fields of the SQLCA.

BEFORE

Positions the cursor before the first row of the result table, values are not assigned to host variables.

row-positioned

Positioning of the cursor with row-positioned fetch orientations NEXT, PRIOR, and RELATIVE is done in relation to the current cursor position. Following a successful row-positioned FETCH statement, the cursor is positioned on a single row of data. If the cursor is enabled for rowsets, positioning is performed relative to the first row of the current rowset, and the cursor is positioned on a rowset consisting of a single row.

NEXT is the only row-positioned fetch operation that can be explicitly specified for cursors that are defined as NO SCROLL (SQLSTATE 42872, SQLCODE -225).

NEXT
: Positions the cursor on the next row or rows of the result table relative to the current cursor position, and returns data if a target is specified

PRIOR
Positions the cursor on the previous row or rows of the result table relative to the current cursor position, and returns data if a target is specified.

FIRST
For a single row fetch, positions the cursor on the first row of the result table, and returns data if a target is specified.

LAST
For a single row fetch, positions the cursor on the last row of the result table, and returns data if a target is specified.

CURRENT
For single row fetch, the cursor position is not changed, data is returned if a target is specified. If the cursor was positioned on a rowset of more than one row, then the cursor position is on the first row of the rowset.

ABSOLUTE
Data is returned if the specified position is within the rows of the result table, and a target is specified.[4]

RELATIVE
Data is returned if the specified position is within the rows of the result table, and a target is specified.

rowset-positioned

Positioning of the cursor with rowset-positioned fetch orientations NEXT ROWSET, PRIOR ROWSET, and ROWSET STARTING AT RELATIVE is done in relation to the current rowset. The number of rows in the rowset is determined either explicitly or implicitly. The FOR n ROWS clause in the multiple-row-fetch clause is used to explicitly specify the size of the rowset. The rowset size is implicitly set to 1 if the previous FETCH statement did not contain a FOR n ROWS clause, or the previous FETCH statement was a row positioned FETCH statement. Following a successful rowset-positioned FETCH statement, the cursor is positioned on all rows of the rowset.

A rowset-positioned fetch orientation must not be specified if the current cursor position is not defined to access rowsets (SQLSTATE 24523, SQLCODE -249). NEXT ROWSET is the only rowset-positioned fetch orientation that can be specified for cursors that are defined as NO SCROLL (SQLSTATE 42872, SQLCODE -225).

NEXT ROWSET
Positions the cursor on the next rowset of the result table relative to the current cursor position, and returns data if a target is specified. If a row of the rowset reflects a hole, a warning is returned (SQLSTATE 02502, SQLCODE +222), data values are not assigned to host-variable-arrays for that row (i.e., the corresponding positions in the target host-variable-arrays are untouched) and -3 is returned in all provided indicator variables for that row. If a hole is detected, and at least one indicator variable is not provided, then an error is returned (SQLSTATE 24519, SQLCODE -247). If the cursor is not positioned due to a prior error, values are not assigned to the host-variable-array, and an error is returned (SQLSTATE 24513, SQLCODE -227). If a row of the result set would be after the last row of the result table, values are not assigned to host-variable-arrays for that row and any subsequent requested rows of the result set, and a warning is returned (SQLSTATE 02000, SQLCODE +100). NEXT ROWSET is the only rowset positioned fetch orientation that can be explicitly be specified for cursors that are defined as NO SCROLL (SQLSTATE 42872, SQLCODE -225).

4. Please add this PP between the existing 'If K = 0 ...' PP and the 'If an absolute position... PPs for the current explanation for ABSOLUTE.

PRIOR ROWSET

Positions the cursor on the previous rowset of the result table relative to the current position, and returns data if a target is specified. If a row of the rowset reflects a hole, a warning is returned (SQLSTATE 02502, SQLCODE +222), data values are not assigned to host variable arrays for that row (i.e., the corresponding positions in the target host variable arrays are untouched), and -3 is returned in all provided indicator variables for that row. If a hole is detected, and at least one indicator variable is not provided, then an error is returned (SQLSTATE 24519, SQLCODE -247). If the cursor is not positioned due to a prior error, values are not assigned to the host-variable-array, and an error is returned (SQLSTATE 24513, SQLCODE -227). If a row of the result set would be before the first row of the result table, values are not assigned to host-variable-arrays for that row and any requested rows of the result set that logically precede that row, and a warning is returned (SQLSTATE 02000, SQLCODE +100). The prior rowset is logically obtained by fetching the row that precedes the current rowset, and fetching prior rows until the number of rows in the rowset is obtained or the first row of the result table is reached. Although the rowset is logically obtained by fetching backwards from before the current rowset, the data is returned to the application starting with the first row of the rowset, to the end of the rowset.

FIRST ROWSET

Positions the cursor on the first rowset of the result table, and returns data if a target is specified. If a row of the rowset reflects a hole, a warning is returned (SQLSTATE 02502, SQLCODE +222), data values are not assigned to host variable arrays for that row (i.e., the corresponding positions in the target host variable arrays are untouched), and -3 is returned in all provided indicator variables for that row. If a hole is detected, and at least one indicator variable is not provided, then an error is returned (SQLSTATE 24519, SQLCODE -247). If the result table contains fewer rows than the width of the rowset, values are not assigned to host-variable-arrays after the last row of the result table, and a warning is returned (SQLSTATE 02000, SQLCODE +100).

LAST ROWSET

Positions the cursor on the last rowset of the result table and returns data if a target is specified. If a row of the rowset reflects a hole, a warning is returned (SQLSTATE 02502, SQLCODE +222), data values are not assigned to host variable arrays for that row (i.e., the corresponding positions in the target host variable arrays are untouched), and -3 is returned in all provided indicator variables for that row. If a hole is detected, and at least one indicator variable is not provided, then an error is returned (SQLSTATE 24519, SQLCODE -247). If the result table contains fewer rows than the width of the rowset, the last rowset is the same as the first rowset, values are not assigned to host-variable-arrays after the last row of the result table, and a warning is returned (SQLSTATE 02000, SQLCODE +100). The last rowset is logically obtained by fetching the last row of the result table and fetching prior rows until the number of rows in the rowset is obtained or the first row of the result table is reached. Although the rowset is logically obtained by fetching backwards from the bottom of the result table, the data is returned to the application starting with the first row of the rowset, to the end of the rowset.[5]

CURRENT ROWSET

Retains the position of the cursor on the current rowset, and returns data if a target is specified. If a row of the rowset reflects a hole, a warning is returned (SQLSTATE 02502, SQLCODE +222), and data values are not assigned to host-variable-arrays for that row (i.e., the corresponding positions in the target host-variable-arrays are untouched) and -3 is returned in all provided indicator variables for that row. If a hole is detected, and at least one indicator variable is not provided, then an error is returned (SQLSTATE 24519, SQLCODE -247). If the

5. The end of the rowset in this case is also the end of the result table.

cursor is not positioned due to a prior error, values are not assigned to the host-variable-array, and an error is returned (SQLSTATE 24513, SQLCODE -227). If the current rowset contains fewer rows than the width of the cursor then a warning is returned (SQLSTATE 02000, SQLCODE +100).

ROWSET STARTING AT ABSOLUTE or RELATIVE host-variable or integer-constant

Positions the cursor on the rowset beginning at the row of the result table that is indicated by the ABSOLUTE or RELATIVE specification, and returns data if a target is specified.

The indicated host-variable or integer-constant is assigned to an integral value k. If a host-variable is specified, it must be an exact numeric type with scale zero (SQLSTATE 42618, SQLCODE -5012), and must not include an indicator variable (SQLSTATE 42601, SQLCODE -104). The possible data types for the host variable are DECIMAL(n,0), INTEGER, or SMALLINT, where the DECIMAL data type is limited to DECIMAL(19,0) (SQLSTATE 56051, SQLCODE -20127). If a constant is specified, the value must be an integer (SQLSTATE 42601, SQLCODE -104).

ABSOLUTE

If k=0, an error is returned (SQLSTATE 42615, SQLCODE -644). If k>0 then the first row of the rowset is row k. If k<0 then the rowset is positioned on the ABS(k) rows from the bottom of the result table. Assume that ABS(k) is equal to the number of rows in the result table:

- FETCH ROWSET STARTING AT ABSOLUTE -k is the same as FETCH LAST ROWSET.
- FETCH ROWSET STARTING AT ABSOLUTE 1 is the same as FETCH FIRST ROWSET.

RELATIVE

If k=0 and the FOR n ROWS clause does not specify a number different from the number of rows of the current rowset, then the position of the cursor does not change (that is, "RELATIVE ROWSET 0" is the same as "CURRENT ROWSET"). If k=0 and the FOR n ROWS clause specifies a number different from the number of rows of the current rowset, then the cursor is repositioned on the specified number of rows, starting with the first row of the current rowset. Otherwise, RELATIVE repositions the cursor so that the first row of the new rowset cursor position is on the row in the result table that is either k rows after the first row of the current rowset cursor position if k>0, or ABS(k) rows before the first row of the current rowset cursor position if k<0. Assume that ABS(k) is equal to the number of rows for the rowset

- FETCH ROWSET STARTING AT RELATIVE -k is the same as FETCH PRIOR ROWSET.
- FETCH ROWSET STARTING AT RELATIVE k is the same as FETCH NEXT ROWSET.
- FETCH ROWSET STARTING AT RELATIVE 0 is the same as FETCH CURRENT ROWSET.

If a row of the rowset reflects a hole, a warning is returned (SQLSTATE 02502, SQLCODE +222), data values are not assigned to host variable arrays for that row (i.e., the corresponding positions in the target host variable arrays are untouched), and -3 is returned in all provided indicator variables for that row. If a hole is detected, and at least one indicator variable is not provided, then an error is returned (SQLSTATE 24519, SQLCODE -247). If a row of the result set would be after the last row or before the first row of the result table, values are not assigned to host-variable-arrays for that row, and a warning is returned (SQLSTATE 02000, SQLCODE +100).

Row Positioned and Rowset Positioned FETCH Statement Interaction

Table 1 demonstrates the interaction between row positioned and rowset positioned FETCH statements.

For the purposes of this example, assume we have the following:

```
TABLE T1 has 15 rows
```

```
CURSOR CS1 is declared as follows:
```

```
DECLARE CS1 SCROLL CURSOR WITH ROWSET POSITIONING FOR
SELECT * FROM T1;
```

```
an OPEN CURSOR statement has been successfully executed for CURSOR CS1.
```

Assume that the FETCH statements in the Table are executed in the order they appear in the table.

Table 1. Interaction between row positioned and rowset positioned FETCH statements

FETCH Statement	Cursor Position
FETCH FIRST	Cursor is positioned on row 1.
FETCH FIRST ROWSET	Cursor is positioned on a rowset of size 1, consisting of row 1.
FETCH FIRST ROWSET FOR 5 ROWS	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5.
FETCH CURRENT ROWSET	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5.
FETCH CURRENT	Cursor is positioned on row 1
FETCH FIRST ROWSET FOR 5 ROWS	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5.
FETCH or FETCH NEXT	Cursor is positioned on row 2.
FETCH NEXT ROWSET	Cursor is positioned on a rowset of size 1, consisting of row 3.
FETCH NEXT ROWSET FOR 3 ROWS	Cursor is positioned on a rowset of size 3, consisting of rows 4,5, and 6.
FETCH NEXT ROWSET	Cursor is positioned on a rowset of size 3, consisting of rows 7,8, and 9.
FETCH LAST	Cursor is positioned on row 15.
FETCH LAST ROWSET FOR 2 ROWS	Cursor is positioned on a rowset of size 2, consisting of rows 14 and 15.
FETCH PRIOR ROWSET	Cursor is positioned on a rowset of size 2, consisting of rows 12 and 13.
FETCH ABSOLUTE 2	Cursor is positioned on row 2.

Table 1. Interaction between row positioned and rowset positioned FETCH statements (continued)

FETCH Statement	Cursor Position
FETCH ROWSET STARTING AT ABSOLUTE 2 FOR 3 ROWS	Cursor is positioned on a rowset of size 3, consisting of rows 2, 3, and 4.
FETCH RELATIVE 2	Cursor is positioned on row 4.
FETCH ROWSET STARTING AT ABSOLUTE 2 FOR 4 ROWS	Cursor is positioned on a rowset of size 4, consisting of rows 2, 3, 4, and 5.
FETCH RELATIVE -1	Cursor is positioned on row 1.
FETCH ROWSET STARTING AT ABSOLUTE 3 FOR 2 ROWS	Cursor is positioned on a rowset of size 2, consisting of rows 3 and 4.
FETCH ROWSET STARTING AT RELATIVE 4	Cursor is positioned on a rowset of size 2, consisting of rows 7 and 8.
FETCH PRIOR	Cursor is positioned on row 6.
FETCH ROWSET STARTING AT ABSOLUTE 13 FOR 5 ROWS	Cursor is positioned on a rowset of size 3, consisting of rows 13, 14, and 15.
FETCH FIRST ROWSET	Cursor is positioned on a rowset of size 5, consisting of rows 1, 2, 3, 4, and 5. **Note:** Even though the previous FETCH statement returned only 3 rows because EOF was encountered, DB2 will remeber that 5 rows were requested by the previous FETCH statement.

cursor-name

Identifies the cursor to be used in the fetch operation. The cursor name must identify a declared cursor, as explained in the description of the DECLARE CURSOR statement, or an allocated cursor, as explained in "ALLOCATE CURSOR". When the FETCH statement is executed, the cursor must be in the open state (SQLSTATE 24501, SQLCODE -501).

If a single-row-fetch or multiple-row-fetch clause is not specified, the cursor position is adjusted as specified, but no data is returned to the user.

multiple-row-fetch

FOR host-variable or integer ROWS

The indicated host-variable or numeric-constant is assigned to an integral value k. If a host-variable is specified, it must be an exact numeric type with a scale of zero (SQLSTATE 42618, SQLCODE -5012), and must not include an indicator variable (SQLSTATE 42601, SQLCODE -104). Furthermore, k must be in the range, 0<k<=32767 (SQLSTATE 42873, SQLCODE -246). This clause must not be specified for a cursor that is defined without rowset access (SQLSTATE 24518, SQLCODE -20185).

The cursor is positioned on the row specified by the orientation clause (e.g., NEXT ROWSET), and those rows are fetched if a target is specified. Then the next k-1 rows are fetched (moving forward from the cursor position in the result table), until the end of data condition is returned (SQLSTATE 02000, SQLCODE +100), until k-1 rows have been fetched, or until an assignment error or warning is returned. The cursor is positioned on all the rows successfully retrieved. The values from each individual fetch are placed in data areas described in the INTO or USING clause.

See "Considerations for an SQLCA with a FETCH Statement on Page 18" .

INTO host-variable-array

Identifies for each column of the result table a host-variable-array to receive the data retrieved with this FETCH statement. If the number of host-variable-arrays is less than the number of columns of the result table, the SQLQARN3 field of the SQLCA is set to 'W'. Note that there is no warning if there are more host-variable-arrays than the number of columns in the result table.

host-variable-array

Each host-variable-array must be defined in the application program in accordance with the rules for declaring an array (SQLSTATE 42618, SQLCODE -312). See "Application Programming - Host Language Declarations" on page 49 for more information regarding the declarations of host-variable-arrays. A host-variable-array will be used to return the values for a column of the result table. The number of rows to be fetched must be less than or equal to the dimension of each of the host-variable-arrays (SQLSTATE 42873, SQLCODE -246).

An optional indicator array can be specified for a host-variable-array. It should be specified if the SQLTYPE of any SQLVAR occurrence indicates that the column of the result table is nullable. Additionally, if operations that may result in null values, such as UPDATE operations that could cause hole rows, are performed in the application, an indicator arrays should be specified. Otherwise an error will be returned if there are any null values. The indicators are returned as small integers.

INTO DESCRIPTOR descriptor

Identifies an SQLDA that must contain a valid description of zero or more host-variable-arrays or buffers into which the values for a column of the result table are to be returned.

Before the FETCH statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA when processing the statement.
- SQLVAR occurrences to indicate the attributes of an element of the host-variable-array. Within each SQLVAR:
  - SQLTYPE indicates the data type of the elements of the host-variable-array.
  - SQLDATA field points to the first element of the host-variable-array.
  - the length fields (SQLLEN and SQLLONGLEN) are set to indicate the maximum length of a single element of the array.
  - SQLNAME - The fifth through eighth bytes of the data portion of SQLNAME must be initialized to a binary integer representation of the dimension of the host-variable-array. The length of SQLNAME should be set to 8, and the first two bytes of the data portion of SQLNAME must be initialized to X'0000'[6].

The user sets the SQLDATA and SQLIND pointers to the beginning of the corresponding arrays. The SQLDA must have enough storage to contain all SQLVAR

6. The third and forth bytes of the data portion of the SQLNAME field are interpreted as a CCSID if the sixth byte of SQLDAID = '+'.

occurrences (SQLSTATE 07002, SQLCODE -804). Each SQLVAR occurrence describes a host-variable-array or buffer into which the values for a column in the result table are to be returned. If any column of the result table is a LOB then two SQLVAR entries must be provided for each SQLVAR and SQLN must be set to two times the number of SQLVARS. SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

Using FETCH To Retrieve Multiple Rows of Data

A single FETCH statement can be used to retrieve multiple rows of data from the result table of a query. The program controls how many rows are returned on a single FETCH statement by requesting a specific number of rows with the FOR n ROWS clause of the FETCH statement. The maximum number of rows that can be requested on a single FETCH statement is 32767. Once the data is retrieved, the cursor is positioned all the rows retrieved. Fetching stops as soon as an error is returned, all requested rows are fetched or the end of data condition is reached.[7]

Multiple row FETCH statements can be written in C and C++, COBOL, and PL/I [8]. Each of these languages allow the application to code host-variable-arrays and indicator arrays. An indicator array should contain one indicator entry for each row that is fetched.

Fetching multiple rows of data can be done with both serial and scrollable cursors. The operations used to define, open, and close a cursor used for fetching multiple rows of data are the same as for those used for single row FETCH statements. The differences are that a FETCH statement that returns multiple rows of data includes a FOR n ROWS clause to indicate the desired number of rows and a clause to specify the storage where the rows are placed, and the DECLARE CURSOR statement for the cursor that is being fetched from contains the WITH ROWSET POSITIONING clause.

Examples of Fetching Multiple Rows with a Single FETCH Statement.

Givent the cursor C1 defined as:

```
DECLARE C1 CURSOR WITH ROWSET POSITIONING FOR SELECT * FROM EMP;
```

- Fetch the previous rowset, and have the cursor positioned on that rowset.

  ```
  FETCH PRIOR ROWSET FROM C1 FOR 3 ROWS INTO....
  ```

  or

  ```
  FETCH ROWSET STARTING AT RELATIVE -3 FROM C1 FOR 3 ROWS INTO....
  ```

- Fetch 3 rows starting with row 20 regardless of the current position of the cursor, and cause the cursor to be positioned on that rowset at the end of the fetch operation.

  ```
  FETCH ROWSET STARTING AT ABSOLUTE 20 FROM C1 FOR 3 ROWS INTO....
  ```

- Fetch the first x rows and leave the cursor positioned on that rowset at the completion of the fetch.

  ```
  FETCH FIRST ROWSET FROM C1 FOR :x ROWS INTO...
  ```

Considerations for Using the FOR n ROWS Clause with the FETCH FIRST n ROWS ONLY clause

A clause specifying the desired number of rows can be specified in either the SELECT statement of the cursor, or on a FETCH statement for the cursor, or in both. However, these clauses have a different effect:

7. For remote clients, if fetching from a row positioned cursor, multiple rowsets may be returned to the client with one fetch.

8. ASSEMBLER and other languages are supported, but this support is limited to statements that allow USING DESCRIPTOR. The precompiler will not recognize host-variable-arrays exept in C/C++, COBOL, and PL/I.

- in the SELECT statement - a FETCH FIRST n ROWS ONLY clause controls the maximum number of rows that can be accessed with the cursor. When a FETCH statement attempts to retrieve a row beyond the number specified in the FETCH FIRST n ROWS ONLY clause of the SELECT statement then an end of data condition occurs.
- in a FETCH statement - a FOR n ROWS clause controls the number of rows that are returned for a single FETCH statement.

Both of these clauses can be specified.

Diagnostics information for rowset positioned FETCH Statement

The SQLCA is used to return information on errors and warnings found while fetching from a rowset cursor. Processing stops when the end of data is encountered, or when an error (negative SQLCODE) occurs. The specific error or end of data condition is returned as a result of the fetch from the rowset cursor.

After each FETCH statement from a rowset cursor, information is returned to the program through the SQLCA. The SQLCA is set as follows:

- SQLCODE contains the SQLCODE.
- SQLSTATE contains the SQLSTATE.
- SQLERRD3 contains the actual number of rows returned. If SQLERRD3 is less than the number of rows requested, then an error or end-of-data condition occurred.
- SQLWARN flags are set to represent all the warnings that were accumulated while processing the FETCH statement.

Additional information may be obtained about the fetch, including information on all exception conditions encountered while processing the fetch statement, from the GET DIAGNOSTICS statement. See xxx on yyyy for further information.

Consider the following examples, where we attempt to fetch 10 rows with a single FETCH statement.

- **Example 1:** Assume that an error, SQLCODE -802, is detected on the 5th row. SQLERRD3 is set to 4 for the 4 returned rows, SQLSTATE is set to 22003, SQLCODE is set to -802. This information is also available from the GET DIAGNOSTICS STATEMENT, for example:

  ```
  GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
  ```

  ```
  Would result in num_row = 4 and num_cond = 1 (1 condition).
  ```

  ```
  GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
  DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
  ```

  ```
  Would result in sqlstate = 22003, sqlcode = -802, and row_num = 5.
  ```

- **Example 2:** Assume that an end of data condition is detected on the 6th row, and that the cursor does not have immediate sensitivity to updates. SQLERRD3 is set to 5 for the 5 returned rows, SQLSTATE is set to 02000. SQLCODE is set to +100. This information is also available from the GET DIAGNOSTICS STATEMENT, for example:

  ```
  GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
  ```

  ```
  Would result in num_row = 5 and num_cond = 1 (1 condition).
  ```

  ```
  GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
  DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
  ```

  ```
  Would result in sqlstate = 02000, sqlcode = 100, and row_num = 6.
  ```

- **Example 3:** As in Example 2 above, assume that an end of data condition is detected on the 6th row and that the cursor is sensitive to inserted rows. SQLERRD3 is set to 5 for the 5 returned rows, SQLSTATE is set to 02000, and SQLCODE is set to +100. Assume that after the FETCH, 1 more row is inserted into the table. If an additional FETCH statement is executed, an end of data condition is detected after the new row has been fetched. SQLERRD3 is set to 1 for the 1 returned row, SQLSTATE is set to 02000, and SQLCODE is set to +100.

There are some cases where DB2 will return a warning if indicator variables are provided, or an error if indicator variables are not provided. These errors can be thought of as data mapping errors that will result in a warning (SQLCODE +802 for instance) if indicator variables are provided.

- If indicator variables are provided, DB2 returns all rows to the user, marking the errors in the indicator variables. The SQLCODE and SQLSTATE contain the warning from the last data mapping error. The GET DIAGNOSTICS statement may be used to retrieve information about all the data mapping errors that have occured.
- If some or no indicator variables are provided, all rows are returned as above until the first data mapping error is detected which does not have indicator variables. The rows successfully fetched are returned and the SQLSTATE, SQLCODE, and SQLWARN flags are set if necessary. (The SQLCODE may be 0 or a positive value).

It is possible, if a data mapping error occurrs, for the positioning of the cursor to be successful. In this case, the cursor is positioned on the rowset that encountered the data mapping error.

Consider the following examples, which try to fetch 10 rows with a single FETCH statement.

- **Example 1:** Assume that indicators have been provided for values returned for column 1, but not for column 2. The 5th row has a data mapping error (+802) for column 1 and the 7th row has a data mapping error for column 2 (-802 is returned because an indicator was not provided for column 2). SQLERRD3 is set to 6 for the 6 returned rows, SQLSTATE and SQLCODE are set to the error from the 7th row fetched. The indicator variable for the 5th row column 1 indicates a data mapping error was found. This information is also available from the GET DIAGNOSTICS STATEMENT, for example:

  ```
  GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;

  Would result in num_row = 6 and num_cond = 2 (2 conditions).

  GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
  DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;

  Would result in sqlstate = 01519, sqlcode = +802, and row_num = 5.

  GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
  DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;

  Would result in sqlstate = 22003, sqlcode = -802, and row_num = 7.
  ```

  The resulting cursor position is unknown.
- **Example 2:** Assume that null indicators are provided, that rows 3 and 5 are holes, and that data exists for the other requested rows. SQLERRD3 is set to 10 to reflect that 10 fetches were completed, and that information[9] has been returned for 10 rows. SQLSTATE is set to 02502, SQLCODE is set to +222, and all null indicators for rows 3 and 5 are set to -3 to indicate that a hole was detected. This information is also available from the GET DIAGNOSTICS STATEMENT, for example:

9. Note that Eight rows actually contain data. For two rows, indicator variables were set to indicate no data was returned for those rows.

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
```

```
Would result in num_row = 10 and num_cond = 2 (2 conditions).
```

```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
```

```
Would result in sqlstate = 02502, sqlcode = +222, and row_num = 3.
```

```
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
```

```
Would result in sqlstate = 02502, sqlcode = +222, and row_num = 5.
```

If a null indicator was not provided for any variable in a row that was a hole, then an error would be returned (SQLSTATE 24519, SQLCODE -247).

SQLCA Usage Summary: For multiple-row-fetch, the fields of the **SQLCA** are set as follows:

Condition		**Action: Resulting Values Stored in the SQLCA Fields**		
Errors	**DATA**	**SQLSTATE**	**SQLCODE**	**SQLERRD3**
No(1)	Return all requested rows	00000	0	# rows requested
No(1)	Return data for subset of requested rows, End of Data	02000	+100	#rows
No(1)	Return all requested rows	sqlstate(2)	sqlcode(2)	#rows requested
Yes(1)	Return successfully fetched rows	sqlstate(3)	sqlcode(3)	#rows
Yes(1)	Return successfully fetched rows	sqlstate(4)	sqlcode(4)	#rows

Table notes:

(1) SQLWARN flags may be set in all cases, even if there are no other warnings or errors indicated. The warning flags are an accumulation of all warning flags set while processing the multiple-row-fetch.

(2) sqlcode is the last positive SQLCODE, and sqlstate is the corresponding SQLSTATE value.

(3) Database Server detected error. sqlcode is the first negative SQLCODE encountered, sqlstate is the corresponding SQLSTATE value

(4) Client detected error. sqlcode is the first negative SQLCODE encountered, sqlstate is one of the following SQLSTATEs: 22002, 22008, 22509, 22518, or 55021.

Providing indicator variable for error condition: If an error occurs as the result of an arithmetic expression in the SELECT list of an outer SELECT statement (division by zero, or overflow) or a numeric conversion error occurs, the result is the null value. As in any other case of a null value, an indicator variable must be provided and the main variable is unchanged. In this case, however, the indicator variable is set to -2. Processing of the statement continues as if the error had not occurred. (However, this error causes a positive SQLCODE.) If you do not provide an indicator variable, a negative value is returned in the SQLCODE field of the SQLCA. Processing of the statement terminates when the error is encountered. No value is assigned to the host variable or to later variables, though any values that have already been assigned to variables remain assigned. Additionally, a -3 is returned in all indicators provided by the application, when a hole was detected for the row on a rowset positioned FETCH, and values were not returned for the row. Processing of the statement terminates if a hole is detected and at least one indicator variable was not provided by the application (SQLSTATE 24519, SQLCODE - 247).

Examples

Example 1: Fetching the Last 5 Rows of the Result Table

Fetch the last 5 rows of the result table using cursor C1:

```
FETCH ROWSET STARTING AT ABSOLUTE -5

FROM C1 FOR 5 ROWS INTO DESCRIPTOR :MYDESCR;
```

Example 2: multiple-row-fetch Using host-variable-arrays

Fetch 6 rows starting at row 10 for cursor CURS1, and fetch the data into three host-variable-arrays:

```
FETCH ROWSET STARTING AT ABSOLUTE 10

   FROM  CURS1 FOR  6 ROWS

   INTO  :hav1, :hva2, :hva3;
```

Alternatively, a descriptor could have been specified in an INTO DESCRIPTOR clause where the information in the SQLDA reflects the data types of the host-variable-arrays:

```
FETCH ROWSET STARTING AT ABSOLUTE 10

   FROM  CURS1 FOR  6 ROWS

   INTO DESCRIPTOR descriptor-name;
```

Example 3: multiple-row-fetch - Using a host-variable-array

Consider a cursor with two columns, one INTEGER (nullable), and one FLOAT (not nullable). A program could fetch 3 such rows at once with some code like the following C:

```
#include "stdlib.h"

main()

{  EXEC SQL

      INCLUDE SQLCA;

   EXEC SQL

      INCLUDE SQLDA;


   long int cola[20];

   float colb[20];

   short inda[20], indb[20];

   long int Num_of_rows = 3;

   long int i;

   EXEC SQL

      DECLARE C1 SCROLL CURSOR WITH ROWSET POSITIONING FOR

          SELECT A, B FROM T;

   EXEC SQL OPEN C1;

   /* 2 variables in each row */
```

```
    EXEC SQL

     FETCH NEXT ROWSET FROM C1

     FOR :Num_of_rows ROWS

     INTO :cola:inda, :colb:indb;

   if (SQLCODE >= 0 ) {        /* no errors */

      {

      EXEC SQL GET DIAGNOSTICS :num_rows = ROW_COUNT;

      for (i=1;i < num_rows;i++)

       {

       ;/* do something with fetch results  */

       }

      }

   else
      {

      EXEC SQL GET DIAGNOSTICS :num_cond = NUMBER;

      for (i=0;i < num_cond;i++)

       {

      EXEC SQL GET DIAGNOSTICS CONDITION :i
               :sqlstate = RETURNED_SQLSTATE,
               :sqlcode = DB2_RETURNED_SQLCODE,
               :row_num = DB2_ROW_NUMBER;
       printf
        ("SQL error occurred,
           SQLCODE = %d, SQLSTATE = %s, ROW_NUMBER = %d/n",
           sqlcode, sqlstate, row_num);

       }

      }

 done:

   exit();}
```

DRDA Considerations

DB2 UDB for z/OS will limit the size of user data and control information to 10M (except for LOBs which are processed in a different data stream) for a single multiple row FETCH statement using host variable arrays (SQLSTATE 57011, SQLCODE -904).

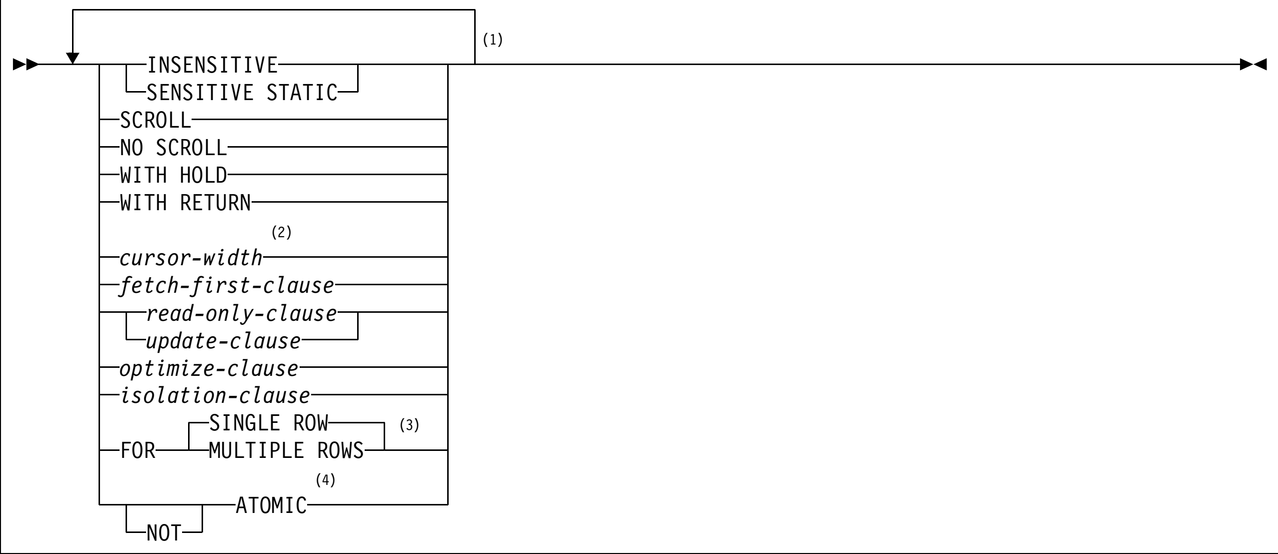
WITH and WITHOUT ROWSET POSITIONING refer to how data is fetched at the server. Data may be blocked, according to existing rules and restrictions for data fetched using either positioning form.

For remote clients, at least one rowset is always be returned in a network request. Multiple rowsets may be returned in a single network request subject to existing block fetch rules and restrictions.

Multi-row insert and fetch statements are supported by any requester or server that supports the DRDA Version 3 protocols. SQLCODE -30005, SQLSTATE 56702 will be returned if an attempt is made to issue a multi-row insert or fetch statement on a server that does not support DRDA Version 3 prtocols.

PREPARE Statement

attribute-string



Note:

1. The same clause must not be specified more than once. If the options are not specified, their defaults are whatever was specified for the corresponding option in an associated DECLARE CURSOR statement.

2. See the **DECLARE CURSOR Statement** for more information about the *cursor-width* clause.

3. The **FOR SINGLE ROW** or **FOR MULTIPLE ROWS** clause must only be specified for an INSERT statement (SQLSTATE 07501, SQLCODE -20186).

4. The **ATOMIC** or **NOT ATOMIC** clause must only be specified for an INSERT statement (SQLSTATE 07501, SQLCODE -20186).

Figure 4. PREPARE Statement

Description

cursor-width: WITHOUT ROWSET POSITIONING or **WITH ROWSET POSITIONING**

See “DECLARE CURSOR Statement” on page 6 for a description of these clauses

SCROLL or NO SCROLL

Specifies whether the cursor is scrollable.

SCROLL
Specifies that the cursor is scrollable. For a scrollable cursor, whether the cursor has sensitivity to inserts, updates, or deletes depends on the cursor sensitivity option in effect for the cursor. If SCROLL is specified and neither ASENSITIVE nor SENSITIVE is specified, then the cursor is read-only and behaves as INSENSITIVE.

NO SCROLL
Specifies that the cursor is not scrollable. This is the default.

FOR SINGLE ROW or FOR MULTIPLE ROWS

Specified whether a variable number of rows of data will be provided for a dynamic INSERT statement.

FOR MULTIPLE ROWS
Specifies that multiple rows of data may be provided with host variable arrays on an EXECUTE statement for the statement being prepared. FOR MULTIPLE ROWS must only be specified for an INSERT statement (SQLSTATE 07501, SQLCODE -20186).

FOR SINGLE ROW
Specifies that multiple rows of data may **not** be provided with host variable arrays on an EXECUTE statement for the statement being prepared. FOR SINGLE ROW must only be specified for an INSERT statement (SQLSTATE 07501, SQLCODE -20186).

ATOMIC or NOT ATOMIC

Specifies whether all of the rows should be inserted as an atomic operation or not. This clause is only valid for a static INSERT statement (SQLSTATE 42601, SQLCODE -104). This clause cannot be used with an INSERT statement that is dynamically executed, see “PREPARE Statement” on page 24 for more information.

Specifies whether all of the rows should be inserted as an atomic operation or not.

ATOMIC

Specifies that if the insert for any row fails, then all changes made to the database by any of the inserts, including changes made by successful inserts, are undone. This is the default.

NOT ATOMIC

Specifies that, regardless of the failure of any particular insert of a row, the EXECUTE statement will not undo any changes made to the database by the successful inserts of other rows from the host-variable-arrays, and INSERTs will be attempted for subsequent rows. However, the minimum level of atomicity is at least that of a single insert (i.e., it is not possible for a partial insert to complete), including any triggers that may have been executed as a result of the insert statement.

INSERT Statement

The INSERT statement inserts rows into a table or view. The table or view can be at the current server or any DB2 subsystem with which the current server can establish a connection. Inserting a row into a view also inserts the row into the table on which the view is based.

There are three forms of this statement:

- The INSERT via VALUES is used to insert a single row into the table or view using the values provided or referenced.
- The INSERT via SELECT is used to insert one or more rows into the table or view using values from other tables, or views, or both.
- The INSERT via FOR n ROWS form is used to insert multiple rows into the table or view using values provided in the *host-variable-array*

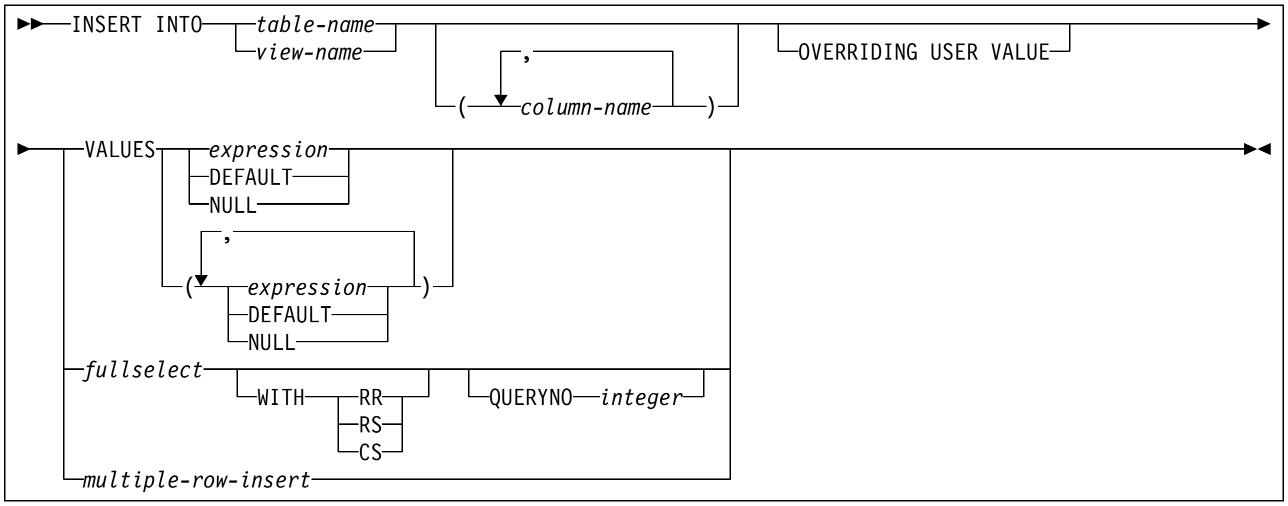


Figure 5. INSERT Statement

multiple-row-insert

```
                                                            ┌─,─────────────────────┐
                                                            ▼                       │
►►─┬──────────────────────────────────────────────┬─┬─VALUES──(────host-variable-array─┴──)─┬──────────►
   │                                        (1)   │ └─USING DESCRIPTOR──descriptor-name──────┘
   └─┬─────┬─┬─integer-constant─┬──ROWS──────────┘
     └─FOR─┘ └─host-variable────┘

   ┌─ATOMIC─────────┐
►──┼────────────────┼───────────────────────────────────────────────────────────────────────────►◄
   │            (2) │
   └─NOT ATOMIC─────┘
```

Note:

1. The FOR n ROWS clause must be specified for a static multiple-row-insert. However, this clause must not be specified for a dynamic INSERT statement (SQLSTATE 42601, SQLCODE -104). For a dynamic statement, the FOR n ROWS clause is specified on the EXECUTE statement.

2. The ATOMIC or NOT ATOMIC clauses may be specified for a static multiple-row-insert. However, this clause must not be specified for a dynamic INSERT statement (SQLSTATE 42601, SQLCODE -104). For a dynamic statement, the ATOMIC or NOT ATOMIC clause is specified as an attribute on the PREPARE statment.

Figure 6. INSERT - multiple-row-insert

multiple-row-insert

FOR integer-constant or **host-variable ROWS**

Specifies the number of rows to be inserted. This clause is only valid for a static INSERT statement (SQLSTATE 42601, SQLCODE -104). For a dynamic INSERT statement, refer to the"EXECUTE Statement" on page 32 for more information.

Evaluates host-variable or integer-constant to an integral value k. If a host-variable is specified, it must be an exact numeric type with scale zero (SQLSTATE 42618, SQLCODE -5012), and must not include an indicator variable (SQLSTATE 42601, SQLCODE -104). Furthermore, k must be in the range, 0<k<=32767 (SQLSTATE 42873, SQLCODE -246). k rows are inserted into the target table from the specified host-variable-array. This form of the ROWS clause can not be used with an INSERT statement that is dynamically executed (SQLSTATE 42601, SQLCODE -104).

VALUES (host-variable-array, ...)

Identifies for each column involved in the INSERT a host variable array that contains the data to be inserted. The number of columns implicitly or explicitly specified in the INSERT statement must be less than or equal to the total number of host-variable-arrays specified (SQLSTATE 07001, SQLCODE -313). Each host variable array must be defined in the application program in accordance with the rules for declaring an array (SQLSTATE 42618, SQLCODE -312). A host-variable-array contains data for the column of the table that is a target of the INSERT. The number of rows to be inserted must be less than or equal to the dimension of each of the host-variable-arrays (SQLSTATE 42873, SQLCODE -246).

An optional indicator array can be specified for each host-variable-array. It should be specified if the SQLTYPE of any SQLVAR occurrence indicates that the SQLVAR is nullable. The indicators must be small integers. The indicator array must be large enough to contain an indicator for each row of input data(SQLSTATE 42873, SQLCODE -246).

USING DESCRIPTOR descriptor

Identifies an SQLDA that must contain a valid description of the host-variable-arrays or buffers which contain the values to be inserted.

The SQLDA must have enough storage to contain a SQLVAR for each target column for which values are provided, plus an additional SQLVAR entry for use by DB2 UDB for z/OS (SQLSTATE 07002, SQLCODE -804). Each SQLVAR occurrence, other than the last one, describes a host variable array or buffer in which the values for a column of the target table are. An additional SQLVAR entry (i.e., the last one) must be provided for use by DB2 UDB for z/OS. For example, if the INSERT statement is providing values for 5 columns of the target table, then 6 SQLVAR entries must be provided. If any value is a LOB then twice as many SQLVAR entries must be provided, and SQLN must be set to that number. So, if the INSERT statement is providing values for 5 columns of the target table, and some of the values to be inserted are LOBs, then 12 SQLVAR entries must be provided.

Before the multiple row INSERT statement is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA that provide values for columns that are the target of the INSERT, plus one.[10]
- SQLVAR occurrences to indicate the attributes of an element of the host variable array for the SQLVAR entries that correspond to values provided for the target columns of the INSERT. Within each SQLVAR:
  - SQLTYPE to indicate the data type of the elements of the host-variable-array
  - SQLDATA field to point to the corresponding host-variable-array
  - the length fields (SQLLEN and SQLLONGLEN) set to indicate the maximum length of a single element of the array.
- SQLNAME - The fifth through eighth bytes of the data portion of SQLNAME must be initialized to a binary integer representation of the dimension of the host-variable-array. The length of SQLNAME should be set to 8, and the first two bytes of the data portion of SQLNAME must be initialized to X'0000'[11].

The user sets the SQLDATA and SQLIND pointers to the beginning of the corresponding arrays. SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

ATOMIC or NOT ATOMIC

Specifies whether all of the rows should be inserted as an atomic operation or not.

ATOMIC

10.The extra entry is for use by DB2 UDB for z/OS to specify the number of rows to be inserted.

11. The third and forth bytes of the data portion of the SQLNAME field are interpreted as a CCSID if the sixth byte of SQLDAID = '+'.

Specifies that if the insert for any row fails, then all changes made to the database by any of the inserts, including changes made by successful inserts, are undone. This is the default.

NOT ATOMIC

Specifies that, regardless of the failure of any particular insert of a row, the INSERT statement will not undo any changes made to the database by the successful inserts of other rows from the host-variable-arrays, and INSERTs will be attempted for subsequent rows. However, the minimum level of atomicity is at least that of a single insert (i.e., it is not possible for a partial insert to complete), including any triggers that may have been executed as a result of the INSERT statement.

Considerations for an SQLCA with an INSERT Statement for Multiple Rows of Data

When NOT ATOMIC is specified the inserts are processed independently. This means that if one or more errors occur during the execution of an INSERT of a row, then processing continues. The row that was being inserted at the time of the error is not inserted. Execution continues with the next row to be inserted, and any other changes made during the execution of the multiple row INSERT statement are not backed out. However, the insert of an individual row is an atomic action.

Otherwise, when ATOMIC is in effect, if an insert value violates any constraints, or if any other error occurs during the execution of an INSERT of a row, then all changes made during the execution of the multiple row INSERT statement are backed out. The SQLCA reflects the last warning encountered.

In either case, after an INSERT statement that inserts multiple rows of data, information is returned to the program through the SQLCA. The SQLCA is set as follows:

- SQLCODE contains the SQLCODE.
- SQLSTATE contains the SQLSTATE
- SQLERRD3 contains the number of rows actually inserted. SQLERRD3 is the number of rows inserted, if this is less than the number of rows requested, then an error occurred.
- SQLWARN flags are set if they were set during any single insert operation.

The SQLCA is used to return information on errors and warnings found during a multiple-row-insert. If indicator arrays are provided, the indicator variable values are used to determine if the value from the host-variable-array, or NULL, will be used. The SQLSTATE contains the warning from the last data mapping error.

Additionally, when NOT ATOMIC is in effect then status information is available for each failure or warning that occurred while processing the insert. The status information for each row is available via the GET DIAGNOSTICS statement. See "GET DIAGNOSTICS" on page 34 for more information.

DRDA Considerations

DB2 UDB for z/OS will limit the size of user data and control information to 10M (except for LOBs which are processed in a different data stream) for a single multiple row INSERT statement using host variable arrays (SQLSTATE 57011, SQLCODE -904).

Multi-row insert and fetch statements are supported by any requester or server that supports the DRDA Version 3 protocols. SQLCODE -30005, SQLSTATE 56702 will be returned if an attempt is made to issue a multi-row insert or fetch statement on a server that does not support DRDA Version 3 prtocols.

Example Insert

Example 1: Inserting a Variable Number of Rows Using host-variable-arrays for Column Values.

Assume that the T1 table has 1 column. INSERT a variable (:hv) number of rows of data into the T1 table. The values to be inserted are provided in a host-variable-array (:hva).

```
EXEC SQL INSERT INTO T1 FOR :hv ROWS VALUES (:hva:hvind) ATOMIC;
```

In this example, :hva represents the host-variable-array and :hvind represents the array of indicator variables.

Example 2: Inserting multiple Rows Using host-variable-arrays for Column Values.

Assume that the T2 table has 2 columns, C1 is a SMALL INTEGER column, and C2 is an INTEGER column. INSERT 10 rows of data into the T2 table. The values to be inserted are provided in host-variable-arrays :hva1 (an array of INTEGERS and :hva2 an array of DECIMAL(15,0) values. The data values for :hva1 and :hva2 are represented in the following table:

Table 2. Data values for :hva1 and :hva2

Array Entry	:hva1	:hva2
1	1	32768
2	-12	90000
3	79	2
4	32768	19
5	8	36
6	5	24
7	400	36
8	73	4000000000
9	-200	2000000000
10	35	88

```
EXEC SQL INSERT INTO T2 (C1, C2) FOR 10 ROWS VALUES (:hva1:hvind1,
:hva2:hvind2) NOT ATOMIC;
```

After execution of the INSERT statement, we will have the following in the SQLCA:

```
SQLCODE = 0
```

```
SQLSTATE = 0
```

```
SQLERRD3 = 8
```

Although we attempted to insert 10 rows, only 8 rows of data were inserted. Further information can be found by using the GET DIAGNOSTICS statement, for example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
```

```
Would result in num_row = 8 and num_cond = 2 (2 conditions).
```

```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
```

```
Would result in sqlstate = 22003, sqlcode = -302, and row_num = 4
```

```
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
```

```
Would result in sqlstate = 22003, sqlcode = -302, and row_num =  8
```

Example 3: Inserting multiple Rows Using host-variable-arrays for Column Values.

Assume the above table T2, with 2 columns, C1 is a SMALL INTEGER column, and C2 is an INTEGER column. INSERT 8 rows of data into the T2 table. The values to be inserted are provided in host-variable-arrays :hva1 (an array of INTEGERS and :hva2 an array of DECIMAL(15,0) values. The data values for :hva1 and :hva2 are represented in Table 2 on page 30.

```
EXEC SQL INSERT INTO T2 (C1, C2) FOR 8 ROWS VALUES (:hva1:hvind1,
:hva2:hvind2) NOT ATOMIC;
```

After execution of the INSERT statement, we will have the following in the SQLCA:

```
SQLCODE = -302
```

```
SQLSTATE = 22003
```

```
SQLERRD3 = 6
```

Although we attempted to insert 8 rows, only 6 rows of data were inserted. Further information can be found by using the GET DIAGNOSTICS statement, for example:

```
GET DIAGNOSTICS :num_rows = ROW_COUNT, :num_cond = NUMBER;
```

```
Would result in num_row = 6 and num_cond = 2 (2 conditions).
```

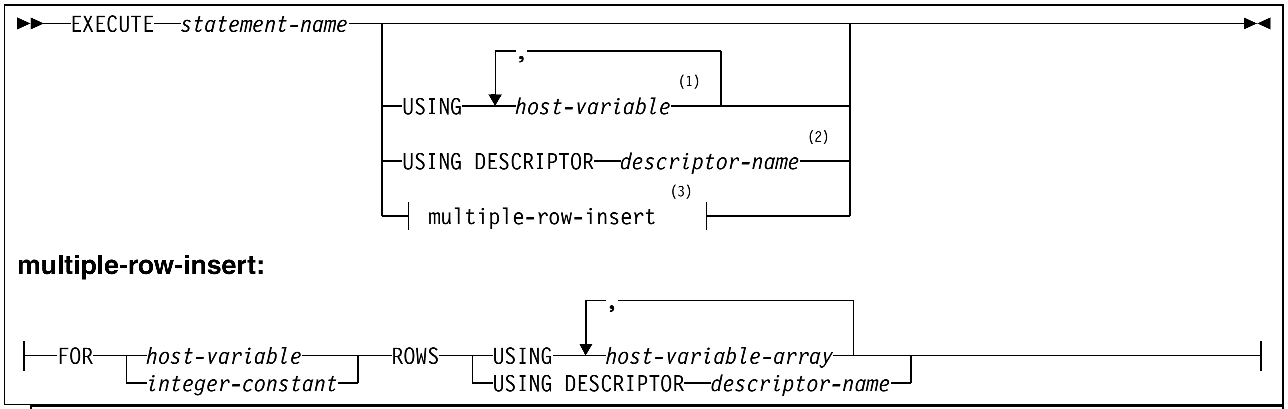
```
GET DIAGNOSTICS CONDITION 1 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
```

```
Would result in sqlstate = 22003, sqlcode = -302, and row_num = 4
```

```
GET DIAGNOSTICS CONDITION 2 :sqlstate = RETURNED_SQLSTATE, :sqlcode =
DB2_RETURNED_SQLCODE, :row_num = DB2_ROW_NUMBER;
```

```
Would result in sqlstate = 22003, sqlcode = -302, and row_num =  8
```

EXECUTE Statement



Note:

1. The FOR n ROWS clause can not be specified with this variation of the EXECUTE Statement.
2. The FOR n ROWS clause can not be specified with this variation of the EXECUTE Statement.
3. This form can only be specified for a dynamic INSERT statement.

Figure 7. EXECUTE STATEMENT

For host-variable or integer ROWS

Specifies the number of rows to be inserted. The values inserted are specified in the USING clause.

host-variable or integer-constant is assigned to an integral value k. If a host-variable is specified, it must be an exact numeric type with a scale of zero (SQLSTATE 42618, SQLCODE -5012), and must not include an indicator variable (SQLSTATE 42601, SQLCODE -104). Furthermore, k must be in the range, 0<k<=32767 (SQLSTATE 42873, SQLCODE -246).

USING host-variable-array, ...

Identifies for each column involved in the INSERT a host-variable-array that contains the data to be inserted. The number of columns implicitly or explicitly specified in the INSERT statement must be less than or equal to the total number of host-variable-arrays specified (SQLSTATE 07001, SQLCODE -313). Each host-variable-array must be defined in the application program in accordance with the rules for declaring an array (SQLSTATE 42618, SQLCODE -312). The number of rows to be inserted must be less than or equal to the dimension of each of the host-variable-arrays (SQLSTATE 42873, SQLCODE -246).

An optional indicator array can be specified for a host-variable-array. It should be specified if the SQLTYPE of any SQLVAR occurrence indicates that the value for the column is null. The number or elements in the indicator array must be greater than or equal to the number of rows being inserted (DSNH5011I).

USING DESCRIPTOR descriptor

Identifies an SQLDA that must contain a valid description of the host-variable-arrays or buffers containing the values to be inserted.

The SQLDA must have enough storage to contain a SQLVAR for each target column for which values are provided, plus an additional SQLVAR entry for use by DB2 UDB for z/OS (SQLSTATE 07002, SQLCODE -804). The precompiler will generate code to fill in the required information for this extra SQLVAR. Each SQLVAR occurrence, other than the last one, describes a host variable array or buffer in which the values for a column of the target table are. An additional SQLVAR entry (i.e., the last one) must be provided for use by DB2 UDB for z/OS. For example, if the INSERT statement is providing values for 5 columns of the target table, then 6 SQLVAR entries must be provided. If any value is a LOB then twice as many SQLVAR entries must be provided, and SQLN must be set to that number. So, if the INSERT statement is providing values for 5 columns of the target table, and some of the values to be inserted are LOBs, then 12 SQLVAR entries must be provided.

Before the dynamic multiple row INSERT is processed, the user must set the following fields in the SQLDA:

- SQLN to indicate the number of SQLVAR occurrences provided in the SQLDA.
- SQLABC to indicate the number of bytes of storage allocated for the SQLDA.
- SQLD to indicate the number of variables used in the SQLDA that provide values for columns that are the target of the INSERT, plus one.
- SQLVAR occurrences to indicate the attributes of an element of the host variable array for the SQLVAR entries that correspond to values provided for the target columns of the INSERT. Within each SQLVAR:
  - SQLTYPE indicates the data type of the elements of the host-variable-array
  - SQLDATA field points to the corresponding host-variable-array
  - the length fields (SQLLEN and SQLLONGLEN) are set to indicate the length of a single element of the array.
- SQLNAME - The fifth through eighth bytes of the data portion of SQLNAME must be initialized to a binary integer representation of the dimension of the host-variable-array. The length of SQLNAME should be set to 8, and the first two bytes of the data portion of SQLNAME must be initialized to X'0000'[12].

The user sets the SQLDATA and SQLIND pointers to the beginning of the corresponding arrays. SQLD must be set to a value greater than or equal to zero and less than or equal to SQLN.

Example

Example 1: Inserting a Variable Number of Rows Using host-variable-arrays for Column Values on EXECUTE

Assume that the IWH.progparm table has 9 columns. Prepare and execute a dynamic INSERT statement which inserts 5 rows of data into the IWH.progparm table. The values to be inserted are provided in arrays, where all the values for a column are provided in an host-variable-array with the EXECUTE statement.

```
stmt = 'INSERT INTO IWH.progparm  (iwhid, updated_by,update_ts,name,
```

12. The third and forth bytes of the data portion of the SQLNAME field are interpreted as a CCSID if the sixth byte of SQLDAID = '+'.

```
                                  short_description, orderNo, parmData,

                                  parmDataLong, VWProgKey)

VALUES ( ? , ? , ? , ? , ? , ? , ? , ? , ? );

attrvar = 'FOR MULTIPLE ROWS';

EXEC SQL PREPARE ins_stmt ATTRIBUTES :attrvar FROM :stmt;

NROWS = 5;

EXEC SQL EXECUTE ins_stmt FOR :NROWS ROWS

           USING :V1, :V2, :V3, :V4, :V5, :V6, :V7, :V8, :V9;
```

In this example, each host variable in the USING clause represents an array of values for the corresponding column of the target of the INSERT statement.

GET DIAGNOSTICS

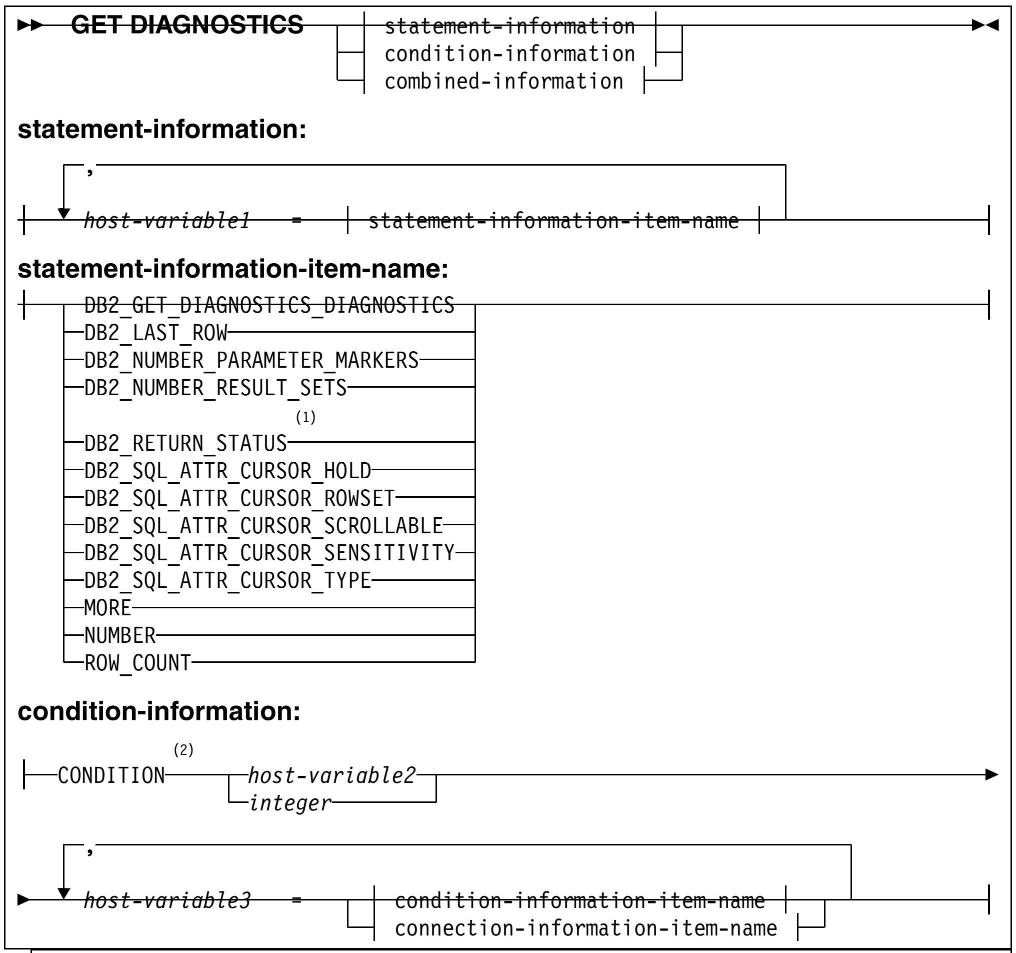
The GET DIAGNOSTIC STATEMENT provides diagnostic information about the last SQL statement (other than a GET DIAGNOSTICS statement) that was executed.

Invocation

This statement can only be embedded in an application program. It is an executable statement that cannot be dynamically prepared.

Authorization

None required.

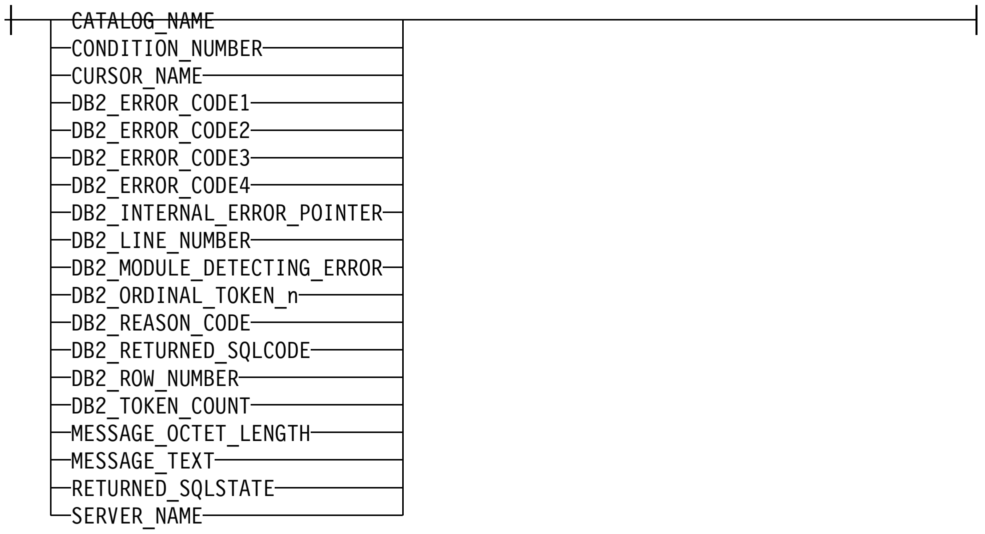


Note:

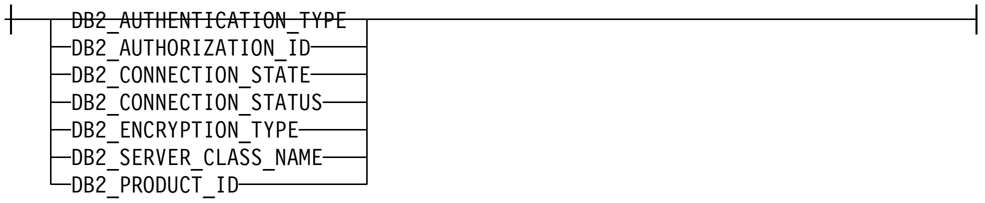
1. RETURN_STATUS can be used as a synonym for DB2_RETURN_STATUS

2. EXCEPTION can be used as a synonym for CONDITION

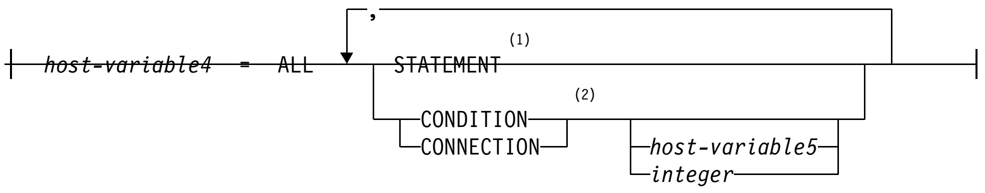
condition-information-item-name:



connection-information-item-name:



combined-information:



Note:

1. STATEMENT can only be specified once.
2. CONDITION and CONNECTION can only be specified once if host-variable5 or integer is not also specified.

Description

The GET DIAGNOSTICS statement is used to gather diagnostic information about the execution of a prior SQL statement. This diagnostic information is gathered as the previous SQL statement is executed, and can logically be thought of as the diagnostics area. Some of the information available via the GET DIAGNOSTICS statement is also available in the SQLCA.

There are three main bodies of information in the diagnostics area. These are the statement-information area, condition-information area, and the combined-information area. After the execution of an SQL statement, there will be information about the execution of the statement in the statement-information area, and there will be at least one instance of the condtion-information area. The number of instances of the condition-information area is indicated by the NUMBER item that is available in the statement-information area. The combined-information area contains a text representation of all the information gathered about the execution of the SQL statement.

statement-information: Information about the last SQL statement Executed.

If a GET DIAGNOSTICS item is not set, then the host variable is set to a default value, based on its data type: 0 for an exact numeric field, an empty string for a VARCHAR field and blanks for a CHAR field.

host-variable1

Identifies a variable described in the program in accordance with the rules for declaring host variables. The data type of the host variable must be the data type specified in Table 1 - "Data Types for GET DIAGNOSTICS" items for the indicated item(SQLCODE -301 and SQLSTATE 42895).

statement-information

DB2_GET_DIAGNOSTICS_DIAGNOSTICS
After a GET DIAGNOSTICS statement, if any errors or warnings occurred in the execution of the GET DIAGNOSTICS statement, contains textual information about these errors or warnings. The format of the information is similar to what would be returned by a GET DIAGNOSTICS :hv = ALL statement.

DB2_LAST_ROW
For a multiple-row FETCH statement, contains a value of +100 if the last row currently in the table is in the set of rows that have been fetched. For cursors that are not sensitive to updates, there would be no need to do a subsequent FETCH, since the result would be an end of data indication (SQLSTATE 02000, SQLCODE=+100). For cursors that are sensitive to updates, a subsequent FETCH may return more data if a row had been inserted before the FETCH was executed. For statements other than multiple-row FETCH statements, or for multiple-row FETCH statements that do not contain the last row, this variable will contain the value 0.

DB2_NUMBER_PARAMETER_MARKERS
For a PREPARE statement, contains the number of parameter markers in the prepared statement. For statements other than PREPARE, or PREPARE statements that contain no paramenter markers, this value will be 0.

DB2_NUMBER_RESULT_SETS
For a CALL statement, contains the actual number of result sets returned by the procedure. For statements other than CALL, or for CALL statements that do not return result sets, this value will be 0.

DB2_RETURN_STATUS
Identifies the status value returned from the previous CALL statement if the procedure was an SQL procedure. If the previous statement is not a CALL statement that invokes an SQL procedure, the value returned will be 0. For more information, see the RETURN statement in DCR DJ031.

DB2_SQL_ATTR_CURSOR _HOLD
For an ALLOCATE or OPEN statement, indicates cursor holdability, whether a cursor can be held open across multiple units of work or not.

- N indicates that this cursor will not remain open across multiple units of work.
- Y indicates that this cursor will remain open across multiple units of work.

Blank Otherwise

DB2_SQL_ATTR_CURSOR_ROWSET
For an ALLOCATE or OPEN statement, indicates rowset accessibility, whether a cursor can be accesses using rowset positioning or not.

- N indicates that this cursor only supports row positioned operations.
- Y indicates that this cursor supports rowset positioned operations.

Blank Otherwise

DB2_SQL_ATTR_CURSOR_SCROLLABLE
For an ALLOCATE or OPEN statement, indicates cursor scrollability, whether a cursor can be scrolled forward and backward or not.

- N indicates that this cursor is not scrollable.
- Y indicates that this cursor is scrollable.

Blank Otherwise

DB2_SQL_ATTR_CURSOR_SENSITIVITY
For an ALLOCATE or OPEN statement, indicates cursor sensitivity, whether a cursor does or does not show updates to cursor rows made by other connections.

- A indicates asensitive.
- I indicates insensitive.
- S indicates sensitive.

Blank Otherwise

DB2_SQL_ATTR_CURSOR_TYPE
For an ALLOCATE or OPEN statement, indicates the type of cursor, whether a cursor type is forward-only, static, or dynamic.

- D indicates a dynamic cursor.
- S indicates a static cursor.

Blank Otherwise

MORE
N indicates that all the warnings and errors from the previous SQL statement were stored in the diagnostic area. **Y** indicates that some of the warnings and errors from the previous SQL statement were discarded because the amount of storage needed to record warnings and errors exceeded 65535 bytes.

NUMBER
Returns the number of errors and warnings detected by the execution of the previous SQL statement into the specified host variable. If the previous SQL statement returned an SQLSTATE of 00000 or no previous SQL statement has been executed, the number returned is one.

ROW_COUNT

Identifies the number of rows associated with the previous SQL statement that was executed.

If the previous SQL statement is a DELETE, INSERT, or UPDATE statement, ROW_COUNT identifies the number of rows deleted, inserted, or updated by that statement, excluding rows affected by either triggers or referential integrity constraints.

If the previous SQL statement is a multiple-row FETCH, ROW_COUNT identifies the number of rows fetched.

If the previous statement is a PREPARE of a SELECT statement, ROW_COUNT identifies the estimated number of result rows in the prepared statement.

For DB2 for z/OS, a value of -1 indicates a mass delete from a table in a segmented table space.

CONDITION:

host-variable2 or **integer**

Identifies the diagnostic for which information is requested. Each diagnostic that occurs while executing an SQL statement is assigned an integer. The value 1 indicates the first diagnostic, 2 indicates the second diagnostic and so on. The host variable specified must be a numeric data type or SQLCODE -301 and SQLSTATE 42895 is returned. An indicator variable is not allowed for this host variable (SQLSTATE 42601, SQLCODE -104). If a value is specified that is less than or equal to zero or greater than the number of available diagnostics, SQLSTATE 35000 and SQLCODE -393 is returned.

host-variable3

Identifies a variable described in the program in accordance with the rules for declaring host variables. The data type of the host variable must be the data type as specified in Table 3 on page 44 for the indicated condition-information item (SQLCODE -301 and SQLSTATE 42895).

condition-information: Assigns the values of the specified condition information to the associated host variables. The host variable specified must be of the data type that is compatible with the data type of the specified diagnostic-id or SQLCODE -301 and SQLSTATE 42895 is returned. If the value of the condition is truncated when assigning it to the host variable, SQLSTATE 01004 and SQLWARN1='W' is returned. If an indicator variable was provided, the length of the value is returned in the indicator variable.

If a DIAGNOSTICS item is not set, then the host variable is set to a default valued, based on the data type of the item. The specific value will be 0 for a numeric field, an empty string for a VARCHAR field, and blanks for a CHAR field.

condition-information-item-name

CATALOG_NAME

If the returned SQLSTATE is:

- class 23 (Integrity Constraint Violation), or
- class 27 (Triggered Data Change Violation), or
- 40002 (Transaction Rollback - Integrity Constraint Violation),

the constraint that caused the error is a referential, check, or unique constraint, the server name of the table that owns the constraint is returned. Otherwise, the empty string is returned.

If the returned SQLSTATE is class 42 (Syntax Error or Access Rule Violation), the server name of the table that caused the error is returned. Otherwise, the empty string is returned.

If the returned SQLSTATE is class 44 (WITH CHECK OPTION Violation), the server name of the view that caused the error is returned. Otherwise, the empty string is returned.

CONDITION_NUMBER

Returns the number of the diagnostic returned.

CURSOR_NAME

If the returned SQLSTATE is class 24 (Invalid Cursor State),the name of the cursor is returned. Otherwise, the empty string is returned.

DB2_ERROR_CODE1[13]

Contains an internal error code. Otherwise, the value 0 is returned.

DB2_ERROR_CODE2[14]

Contains an internal error code. Otherwise, the value 0 is returned.

DB2_ERROR_CODE3[15]

Contains an internal error code. Otherwise, the value 0 is returned.

DB2_ERROR_CODE4[16]

Contains an internal error code. Otherwise, the value 0 is returned.

DB2_INTERNAL_ERROR_POINTER

For some errors, this will be a negative value that is an internal error pointer. Otherwise, the value 0 is returned.

DB2_LINE_NUMBER

For a CREATE PROCEDURE for an SQL procedure where an error is encountered parsing the SQL procedure body, contains the line number where the error was encountered. Otherwise, the value 0 is returned.

DB2_MODULE_DETECTING_ERROR

Contains an identifier indicating which module detected the error. Otherwise, blanks are returned.

DB2_ORDINAL_TOKEN_n

Returns the nth token. n must be a value from 1 to 100. For example, DB2_ORDINAL_TOKEN_1 would return the value of the first token, DB2_ORDINAL_TOKEN_2 the second token, etc... IF there is no value for the token, the empty string is returned.

DB2_REASON_CODE

Contains the reason code for errors that have a reason code token in the message text. Otherwise, the value zero is returned.

DB2_RETURNED_SQLCODE

Returns the SQLCODE for the specified diagnostic.

DB2_ROW_NUMBER

For a statement involving multiple rows, and when such information is available, the row number on which DB2 detected the exception. Otherwise the value 0 is returned.

DB2_TOKEN_COUNT

Returns the number of tokens available for the specified diagnostic id.

13. SQLERRD1
14. SQLERRD2
15. SQLERRD3
16. SQLERRD4

MESSAGE_OCTET_LENGTH

The length of the message text (in bytes).

MESSAGE_TEXT

The message text associated with the SQLCODE.

RETURNED_SQLSTATE

Returns the SQLSTATE for the specified diagnostic.

SERVER_NAME

If the previous SQL statement is a CONNECT, DISCONNECT, or SET CONNECTION, the name of the server specified in the previous statement is returned. Otherwise, the name of the server where the statement executes is returned.

connection-information: Information about the last SQL statement executed if it was a CONNECT.

connection-information-item-name

DB2_AUTHENTICATION_TYPE

Contains an authentication type value of :

- 'S' for a server authentication
- 'C' for client authentication
- 'D' for authentication using DB2 Connect
- 'E' for DCE security services authentication
- blank for unspecified authentication

DB2_AUTHORIZATION_ID

Authorization id used by connected server. Because of userid translation and authorization exits, the local userid may not be the authid used by the server.

DB2_CONNECTION_STATE

Contains a value of -1 if the connection is unconnected; 0 if the connection is local; 1 if the connection is remote. Otherwise, the value zero is returned.

DB2_CONNECTION_STATUS

Contains a value of 1 if committable updates can be performed on the connection for this unit of work; 2 if no committable updates can be performed on the connection for this unit of work. Otherwise, the value 0 is returned.

DB2_SERVER_CLASS_NAME

For a CONNECT or SET CONNECTION statement, contains a value of

- QAS for DB2 UDB for iSeries
- QDB2 for DB2 UDB for OS/390 and z/OS
- QDB2/2 for DB2 UDB for OS/2
- QDB2/6000 for DB2 UDB for AIX
- QDB2/6000 PE for DB2 UDB for AIX Parallel Edition
- QDB2/AIX64 for DB2 UDB for AIX 64-bit
- QDB2/HPUX for DB2 UDB for HP-UX
- QDB2/HP64 for DB2 UDB for HP-UX 64-bit

- QDB2/LINUX for DB2 UDB for Linux
- QDB2/LINUX390 for DB2 UDB for Linux
- QDB2/LINUXIA64 for DB2 UDB for Linux
- QDB2/LINUXPPC for DB2 UDB for Linux
- QDB2/LINUXPPC64 for DB2 UDB for Linux
- QDB2/LINUXZ64 for DB2 UDB for Linux
- QDB2/NT for DB2 UDB for NT
- QDB2/NT64 for DB2 UDB for NT 64-bit
- QDB2/PTX for DB2 UDB for NUMA-Q
- QDB2/SCO for DB2 UDB for SCO UnixWare
- QDB2/SGI for DB2 UDB for Silicon Graphics
- QDB2/SNI for DB2 UDB for Siemens Nixdorf
- QDB2/SUN for DB2 UDB for SUN Solaris
- QDB2/SUN64 for DB2 UDB for SUN Solaris 64-bit
- QDB2/Windows 95 for DB2 UDB for Windows 95 or Windows 98
- QSQLDS/VM for DB2 for VM and VSE
- QSQLDS/VSE for DB2 for VM and VSE

Otherwise, the empty string is returned.

DB2_ENCRYPTION_TYPE
XXXXX

DB2_PRODUCT_ID

Contains a product signature. If the application server is an IBM relational database product, the form is pppvvrrm, where:

- ppp identifies the product as follows:
  - — DSN for DB2 UDB for z/OS
  - — QSQ for DB2 UDB for iSeries
  - — SQL for all other DB2 UDB products
- vv is a two-digit version identifier such as ’08’
- rr is a two-digit release identifier such as ‘01’
- m is a one-digit modification level such as ‘0’

For example, if the application server is Version 8 of DB2 UDB for z/OS, the value would be ‘DSN08010’.

ALL

Indicates that all diagnostic items that are set for the last SQL statement executed should be combined into one string. The format of the string is a semi-colon separated list of all of the available diagnostic information in the form:

<item-name>[(<condition-number>)]=<value-converted-to-character>;...

so for example:

```
NUMBER=1;RETURNED_SQLSTATE=02000;DB2_RETURNED_SQLCODE=+100;
```

host-variable4
Identifies a variable described in the program in accordance with the rules for declaring host variables. The data type of the host variable must be VARCHAR. If the length of host-variable4 is not sufficient to hold the full returned diagnostic string, the string is truncated and no warning or error is given.

STATEMENT
Indicates that all statment-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string. The format is the same as described above for the ALL option.

CONDITION
Indicates that all condition-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string. If host-variable4 or integer is supplied after CONDITION, then the format is the same as described above for the ALL option. If host-variable4 or integer is not supplied, then the format includes a condition number entry at the beginning of the information for that condition in the form:

CONDITION_NUMBER=X;<item-name>=<value-converted-to-character>;... where X is the number of the condition, so for example:

```
CONDITION_NUMBER=1;RETURNED_SQLSTATE=02000;RETURNED_SQLCODE=100;CONDITIO
N_NUMBER=2;RETURNED_SQLSTATE=01004;
```

CONNECTION
Indicates that all connection-information-item-name diagnostic items that are set for the last SQL statement executed should be combined into one string. If host-variable4 or integer is supplied after CONNECTION, then the format is the same as described above for the ALL option. If host-variable4 or integer is not supplied, then the format includes a condition number entry at the beginning of the information for that condition in the form:

CONNECTION_NUMBER=X;<item-name>=<value-converted-to-character>;... where X is the number of the condition, so for example:

```
CONNECTION_NUMBER=1;CONNECTION_NAME=SVL1;DB2_PRODUCT_ID=DSN07010;
```

host-variable5 or integer
Identifies the diagnostic for which ALL CONDITION or ALL CONNECTION information is requested. The host variable specified must be a numeric data type or SQLCODE -301 and SQLSTATE 42895 is returned. An indicator variable is not allowed for this host variable or SQLSTATE 42601 and a product-specific SQLCODE is returned. If a value is specified that is less than or equal to zero or greater than the number of available diagnostics, SQLSTATE 35000 and SQLCODE -393 is returned.

Rules

The GET DIAGNOSTICS statement does not change the contents of the diagnostics area (SQLCA). If an SQLSTATE or SQLCODE special variable is declared in the SQL procedure, these are set to the SQLSTATE or SQLCODE returned from issuing the GET DIAGNOSTICS statement. The SQLSTATE and SQLCODE values from before the GET DIAGNOTICS statement was issued are still available in the diagnostics area by issuing a GET DIAGNOSTICS for RETURNED_SQLSTATE and DB2_RETURNED_SQLCODE.

Examples

Example 1

In an application, use GET DIAGNOSTICS to determine how many rows were updated.

```
long rcount;

EXEC SQL UPDATE T1 SET C1 = C1 + 1;

EXEC SQL GET DIAGNOSTICS :rcount = ROW_COUNT;
```

After execution of this code segment, rcount will contain the number of rows that were updated.

Example 2

In an application, use GET DIAGNOSTICS to handle multiple SQL Errors.

```
long numerrors, counter;

char retsqlstate[5];

long hva[5];

EXEC SQL INSERT INTO T1 FOR 5 ROWS VALUES (:hva) NOT ATOMIC;

EXEC SQL GET DIAGNOSTICS :numerrors = NUMBER;

for ( i=1;i < numerrors;i++)

  {

  EXEC SQL GET DIAGNOSTICS CONDITION :i :retsqlstate = RETURNED_SQLSTATE;

  printf("SQLSTATE = %s",retsqlstate);

  }
```

Execution of this code segment, will set and print retsqlstate with the SQLSTATE for each error that was encountered in the previous SQL statement.

Data Types for GET DIAGNOSTICS Items

Table 3. Data Types for GET DIAGNOSTICS

Item	Data Type
Statement Information	
DB2_GET_DIAGNOSTICS_DIAGNOSTICS	VARCHAR(32672)
DB2_LAST_ROW	INTEGER
DB2_NUMBER_PARAMETER_MARKERS	INTEGER
DB2_NUMBER_RESULT_SETS	INTEGER
DB2_RETURN_STATUS	INTEGER
DB2_SQL_ATTR_CURSOR_HOLD	CHAR(1)
DB2_SQL_ATTR_CURSOR_ROWSET	CHAR(1)
DB2_SQL_ATTR_CURSOR_SCROLLABLE	CHAR(1)
DB2_SQL_ATTR_CURSOR_SENSITIVITY	CHAR(1)
DB2_SQL_ATTR_CURSOR_TYPE	CHAR(1)
MORE	CHAR(1)
NUMBER	INTEGER
ROW_COUNT	DECIMAL(31,0)
Condition Information	

Table 3. Data Types for GET DIAGNOSTICS (continued)

Item	Data Type
CATALOG NAME	VARCHAR(128)
CONDITION_NUMBER	INTEGER
CURSOR_NAME	VARCHAR(128)
DB2_ERROR_CODE1	INTEGER
DB2_ERROR_CODE2	INTEGER
DB2_ERROR_CODE3	INTEGER
DB2_ERROR_CODE4	INTEGER
DB2_INTERNAL_ERROR_POINTER	INTEGER
DB2_LINE_NUMBER	INTEGER
DB2_MODULE_DETECTING_ERROR	CHAR(8)
DB2_ORDINAL_TOKEN_n	VARCHAR(128)
DB2_REASON_CODE	INTEGER
DB2_RETURNED_SQLCODE	INTEGER
DB2_ROW_NUMBER	DECIMAL(31,0)
DB2_TOKEN_COUNT	INTEGER
MESSAGE_OCTET_LENGTH	INTEGER
MESSAGE_TEXT	VARCHAR(32672)
RETURNED_SQLSTATE	CHAR(5)
SERVER_NAME	VARCHAR(128)
Connection Information	
DB2_AUTHENTICATION_TYPE	CHAR(1)
DB2_AUTHORIZATION_ID	VARCHAR(128)
DB2_CONNECTION_STATE	INTEGER
DB2_CONNECTION_STATUS	INTEGER
DB2_ENCRYPTION_TYPE	CHAR(8)
DB2_SERVER_CLASS_NAME	VARCHAR(128)
DB2_PRODUCT_ID	CHAR(8)
Combined Information	
ALL	VARCHAR(32672)

DRDA Considerations

The GET DIAGNOSTICS statement will be supported from a DB2 UDB for z/OS V8 client, regardless of the level of the server (a DB2 UDB for z/OS V7 or a DB2 UDB for

Windows V7 for instance). When connected to servers that do not support the OPEN GROUP Version 3 DRDA standard, a diagnostic condition is generated based on the returned SQLCA. The condition will contain the following information:

- The DB2_RETURNED_SQLCODE is set based on the SQLCODE.
- The RETURNED_SQLSTATE is set based on the SQLSTATE.
- The DB2_GET_DIAGNOSTICS_DIAGNOSTICS item will contain the information from the SQLCA that came from the server.

See "GET DIAGNOSTICS" on page 34 for more information on data items returned from the GET DIAGNOSTICS statement.

Positioned Update

positioned-update

```
►►──UPDATE──┬─table-name─┬──SET──assignment-clause──────────────────────►
            └─view-name──┘

►──┬─WHERE CURRENT OF──cursor-name────────────────────────────────────┬──►◄
   └─FOR CURSOR──cursor-name──FOR ROW──┬─host-variable────┬──OF ROWSET─┘
                                       └─integer-constant─┘
```

Description

WHERE CURRENT OF cursor-name

When the UPDATE statement is executed, the cursor must be positioned on a row or rowset of the result table (SQLSTATE 24504, SQLCODE -508).

- If the cursor is positioned on a single row, that row is the one updated.
- If the cursor is positioned on a rowset then all rows corresponding to the rows of the current rowset are updated.

FOR ROW n OF ROWSET

Specifies which row of the current rowset is to be updated. host-variable or integer-constant is assigned to an integral value k. If a host-variable is specified, it must be an exact numeric type with scale zero (SQLSTATE 42618, SQLCODE -5012), must not include an indicator variable (SQLSTATE 42601, SQLCODE -104), and k must be in the range of 1 to 32767 (SQLSTATE 428B7, SQLCODE -490). The cursor must be positioned on a rowset (SQLSTATE 24520, SQLCODE -589), and the value of host-variable must be a valid value for the set of rows most recently retrieved for the cursor (SQLSTATE 24521, SQLCODE -248).

If the cursor is positioned on a rowset, and the FOR ROW n OF ROWSET clause is not specified then all rows corresponding to the rows of the current rowset are updated.

It is possible for another application process to update a row in the base table of the SELECT statement so that the specified row of the cursor no longer has a corresponding row in the base table. An attempt to update such a row results in an error (SQLSTATE 24510, SQLCODE -222).

Example 1

Assuming Cursor CS1 is positioned on a rowset consisting of 10 rows of table T1, the following UPDATE statement could be used to UPDATE all 10 rows in the rowset:

```
EXEC SQL UPDATE T1 SET C1 = 5 WHERE CURRENT OF CS1;
```

Example 2
Assuming Cusor CS1 is positioned on a rowset consisting of 10 rows of table T1, the following application code could be used to UPDATE the 4th row of the rowset:

```
short ind1, ind2;
int n, updt_value;
stmt = 'UPDATE T1 SET C1 = ? FOR CURSOR CS1 FOR ROW ? OF ROWSET'
ind1 = 0;
ind2 = 0;
n = 4;
updt_value = 5
...
strcpy(my_sqlda.sqldaid,"SQLDA");
my_sqlda.sqln = 2;
my_sqlda.sqld = 2;
my_sqlda.sqlvar[0].sqltype = 497;
my_sqlda.sqlvar[0].sqllen = 4;
my_sqlda.sqlvar[0].sqldata = (int *) &updt_value;
my_sqlda.sqlvar[0].sqlind = (short *) &ind1;
my_sqlda.sqlvar[1].sqltype = 497;
my_sqlda.sqlvar[1].sqllen = 4;
my_sqlda.sqlvar[1].sqldata = (int *) &n;
my_sqlda.sqlvar[1].sqlind = (short *) &ind2;
EXEC SQL PREPARE S1 FROM :stmt;
EXEC SQL EXECUTE S1 USING DESCRIPTOR :my_sqlda;
```

Positioned Delete

positioned-delete

```
►►──DELETE FROM──┬─table-name─┬──────────────────────────────────────────────►
                 └─view-name──┘

►──┬─WHERE CURRENT OF──cursor-name──────────────────────────────────────────►◄
   └─FOR CURSOR──cursor-name──FOR ROW──┬─host-variable─────┬──OF ROWSET─┘
                                       └─integer-constant──┘
```

Description

WHERE CURRENT OF cursor-name
When the DELETE statement is executed, the cursor must be positioned on a row or rowset of the result table (SQLSTATE 24504, SQLCODE -508).

- If the cursor is positioned on a single row, that row is the one deleted, and after the deletion the cursor is positioned before the next row of its result table. If there is no next row, the cursor positioned after the last row.
- If the cursor is positioned on a rowset then all rows corresponding to the rows of the current rowset are deleted, and after the deletion the cursor is positioned before the next rowset of its result table. If there is no next rowset, the cursor positioned after the last rowset.

If the cursor is positioned on a rowset then all rows corresponding to the rows of the current rowset are updated.

FOR ROW n OF ROWSET

Specifies which row of the current rowset is to be deleted. host-variable or integer-constant is assigned to an integral value k. If a host-variable is specified, it must be an exact numeric type with scale zero (SQLSTATE 42618, SQLCODE -5012), must not include an indicator variable (SQLSTATE 42601, SQLCODE -104), and k must be in the range of 1 to 32767 (SQLSTATE 428B7, SQLCODE -490). The cursor must be positioned on a rowset (SQLSTATE 24520, SQLCODE -589), and the value of host-variable must be a valid value for the set of rows most recently retrieved for the cursor (SQLSTATE 24521, SQLCODE -248).

If the cursor is positioned on a rowset, and the FOR ROW n OF ROWSET clause is not specified then all rows corresponding to the rows of the current rowset are deleted.

If the cursor is not positioned on a rowset, the row corresponding to the current position of the cursor is deleted.

It is possible for another application process to delete a row in the base table of the SELECT statement so that the specified row of the cursor no longer has a corresponding row in the base table. An attempt to delete such a row results in an error (SQLSTATE 24510, SQLCODE -222).

Example 1

Assuming Cursor CS1 is positioned on a rowset consisting of 10 rows of table T1, the following DELETE statement could be used to DELETE all 10 rows in the rowset:

```
EXEC SQL DELETE FROM T1 WHERE CURRENT OF CS1;
```

Example 2

Assuming Cusor CS1 is positioned on a rowset consisting of 10 rows of table T1, the following application code could be used to DELETE the 4th row of the rowset:

```
short ind1;

int n;

stmt = 'DELETE FROM T1 FOR CURSOR CS1 FOR ROW ? OF ROWSET'

ind1 = 0;

n = 4;

...

strcpy(my_sqlda.sqldaid,"SQLDA");

my_sqlda.sqln = 1;

my_sqlda.sqld = 1;

my_sqlda.sqlvar[0].sqltype = 497;

my_sqlda.sqlvar[0].sqllen = 4;

my_sqlda.sqlvar[0].sqldata = (int *) &n;

my_sqlda.sqlvar[0].sqlind = (short *) &ind1;

EXEC SQL PREPARE S1 FROM :stmt;

EXEC SQL EXECUTE S1 USING DESCRIPTOR :my_sqlda;
```

Application Programming - Host Language Declarations

Host-variable-arrays may be specified in C/C++, COBOL, and PL/I. In some cases there are specific rules regarding the declaration of host-variable arrays that must be followed or unpredictable results may occur.

For PL/I, the ALIGNED attribute must be specified on arrays of varying length character or varying length graphic variable declarations that are to be used in multi-row SQL.

For C/C++, _Packed attribute may not be specified on the structure declarations for varying length character arrays, varying length graphic arrays, or LOB arrays to be used in multi-row SQL, and the #pragma pack(1) directive may not be in effect if the user is planning to use arrays of varchar, vargraphic, or LOBs in multi-row SQL.

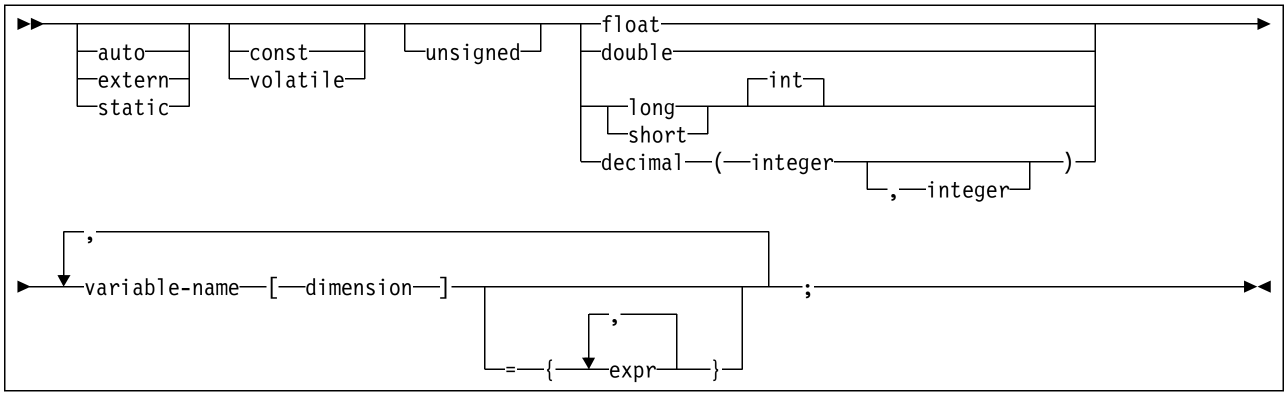
C and C++

Using Host-Variable-Arrays

In C and C++ programs, you can define an array. Host-variable-arrays can be used with the multiple row FETCH and INSERT statements.

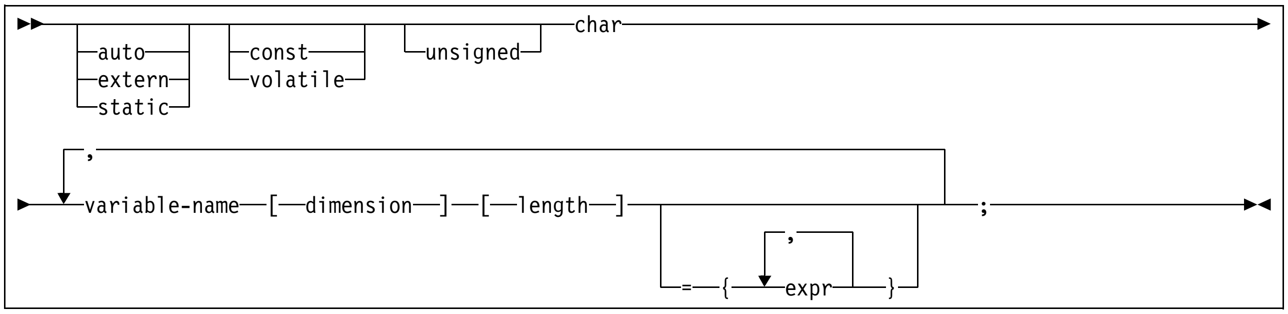
Numeric host-variable-arrays

The following figure shows the syntax for valid numeric host-variable-array declarations.

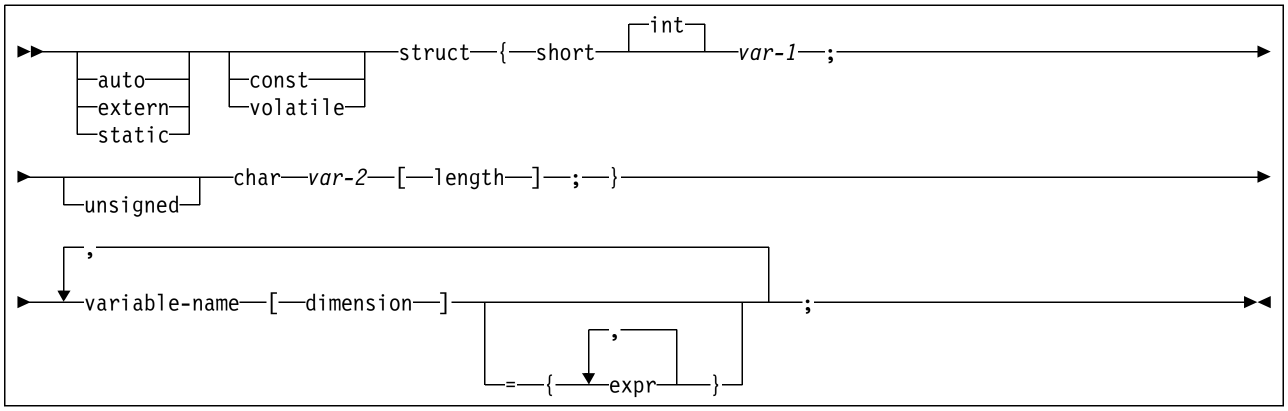


Array of CHARACTER host-variable-arrays (Nul-Terminated)

The following figure shows the syntax for valid character host-variable-array declarations.



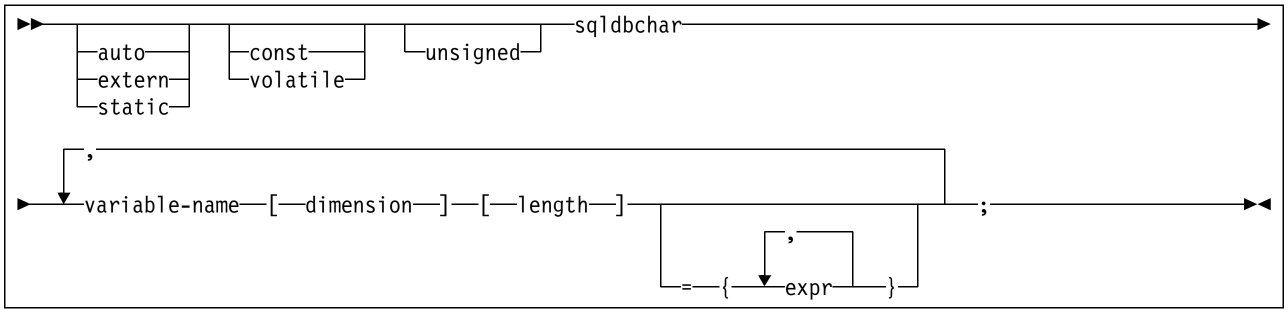
Array of VARCHAR host variables arrays:



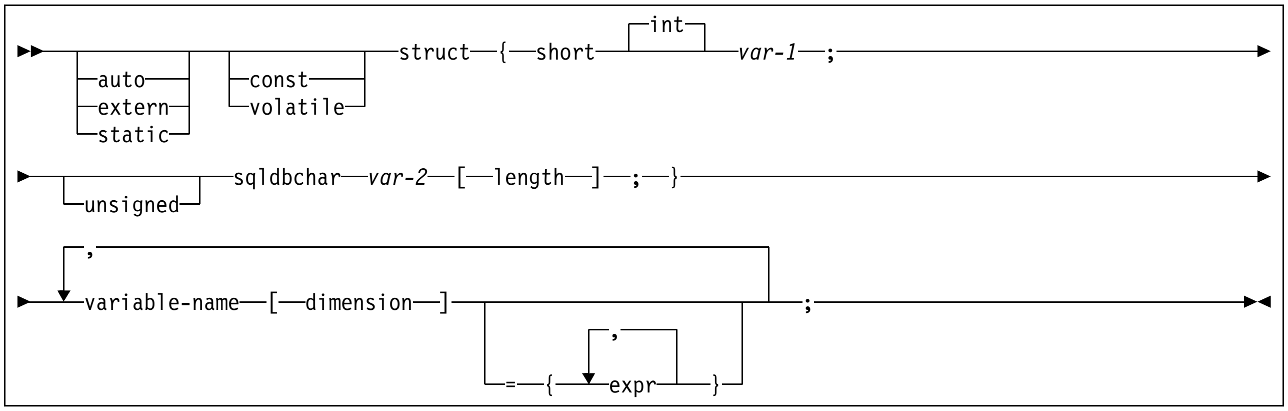
Array of GRAPHIC host-variable-arrays

The following figure shows the syntax for valid GRAPHIC host-variable-array declarations.

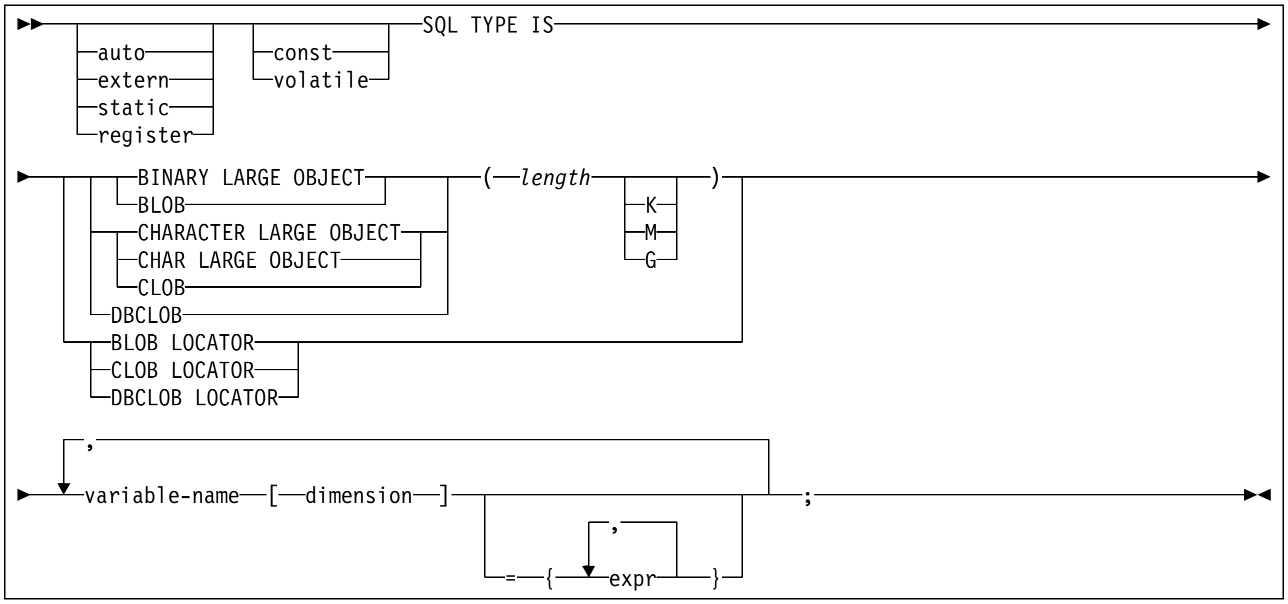
Array of NUL-terminated GRAPHIC variables:



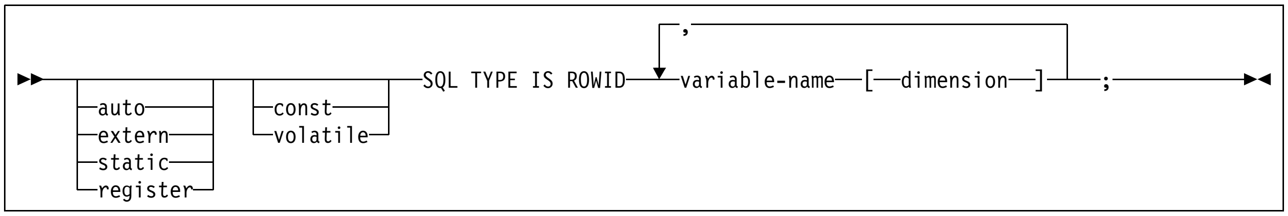
Array of VARGRAPHIC host variables arrays:



LOBs and LOB locator host variables arrays:



ROWID host variables arrays:



Examples

Example 1:

Declare an integer and varying character array to hold columns retrieved from a multi-row fetch statement:

```
long serial_num[10];

struct {

        short len;

        char data[18];

        } name[10]

   ....

EXEC SQL

   DECLARE C1 CURSOR FOR

      SELECT NAME, SERIAL#

        FROM CORPDATA.EMPLOYEE WITH ROWSET POSITIONING;

   ...

EXEC SQL OPEN C1;

EXEC SQL

   FETCH FIRST ROWSET FROM C1 FOR 10 ROWS INTO :name, :serial_num;
```

COBOL

Using host-variable-arrays

In COBOL programs, you can define an array. Host-variable-arrays can be used with the multiple row FETCH and INSERT statements.

Numeric host-variable-arrays

The following figure shows the syntax for valid numeric host-variable-array declarations.

Note: In all of the following, level-1 can be any value from 2 through 48. Also, to save space, it is not specified whether the constant in the VALUES clause is numeric, character, or graphic. However, this information should be specified in the Application Programming and SQL Guide.

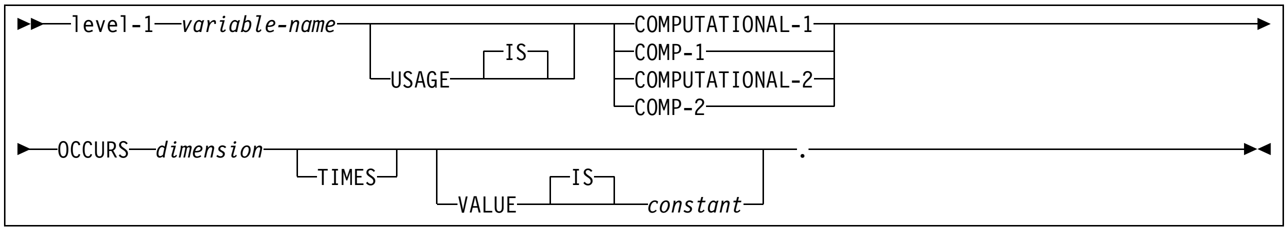


Figure 8. Floating point:

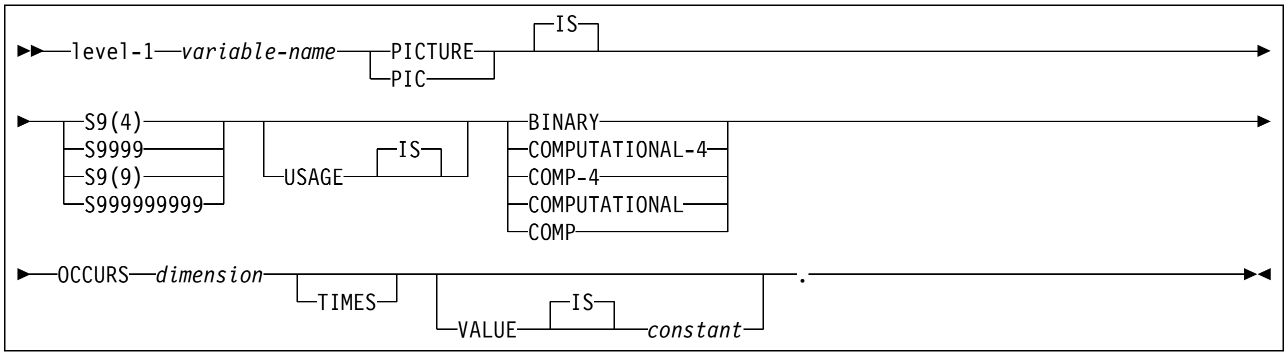


Figure 9. Integer and small integer:

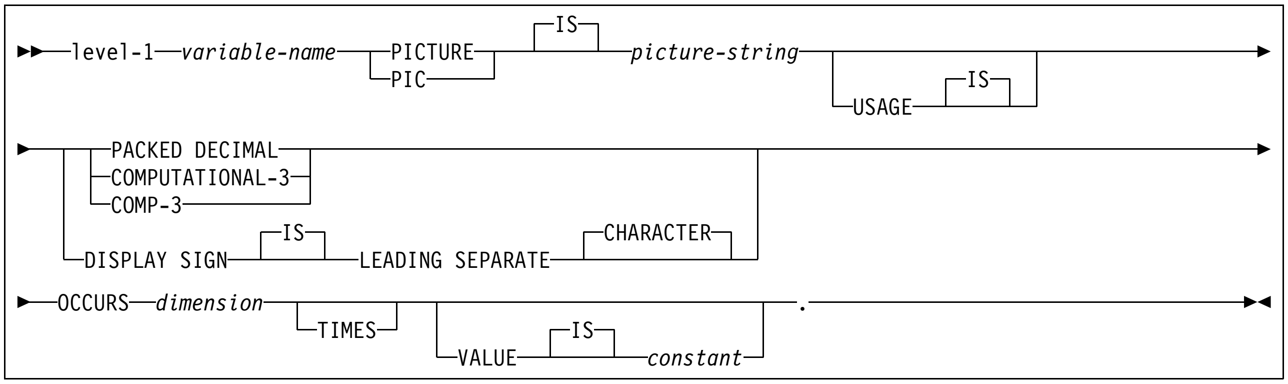


Figure 10. Decimal:

Character string arrays:

The following figure shows the syntax for valid CHARACTER string host-variable-array declarations.

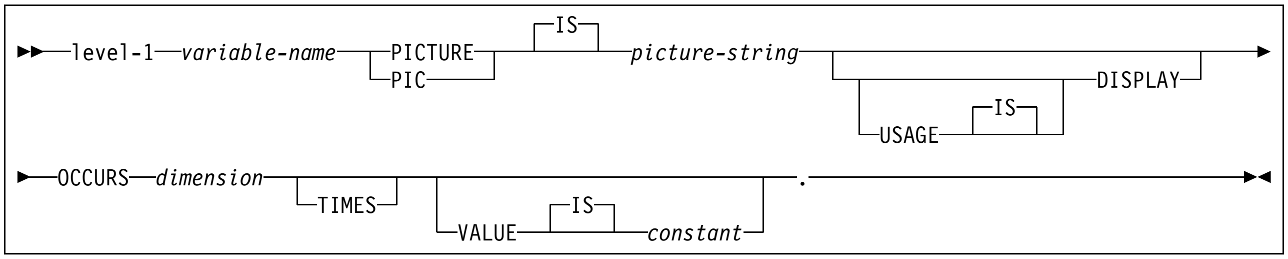


Figure 11. Fixed length character

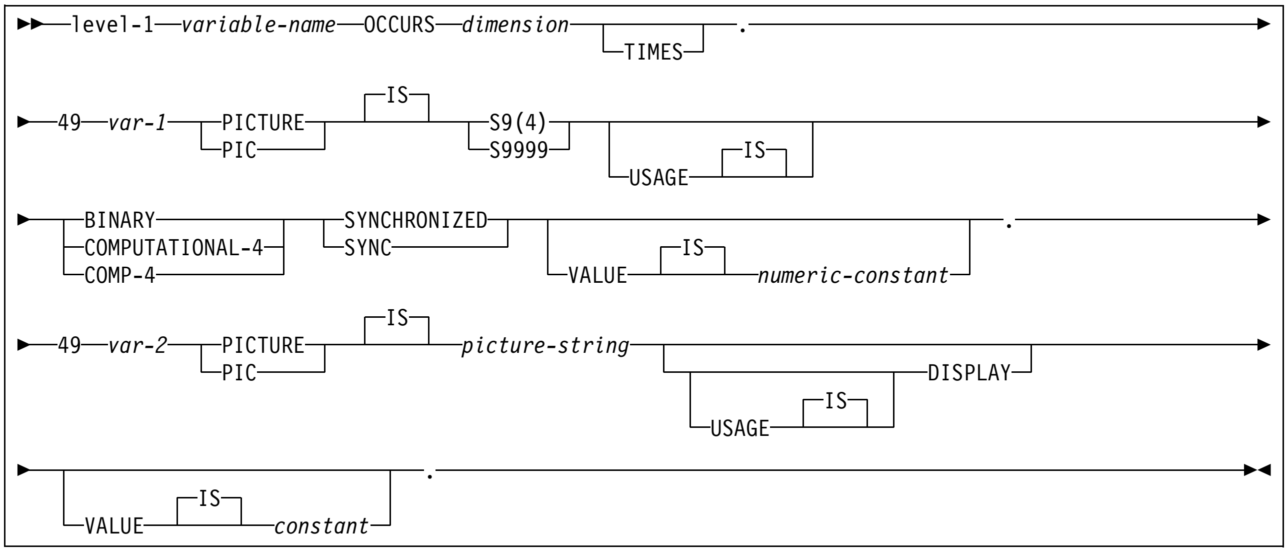


Figure 12. Varying length character

Graphic string arrays:

The following figure shows the syntax for valid GRAPHIC string host-variable-array declarations.

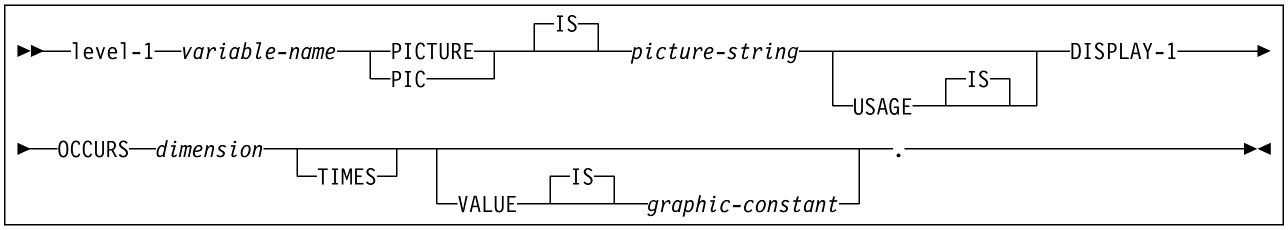


Figure 13. Fixed length graphic

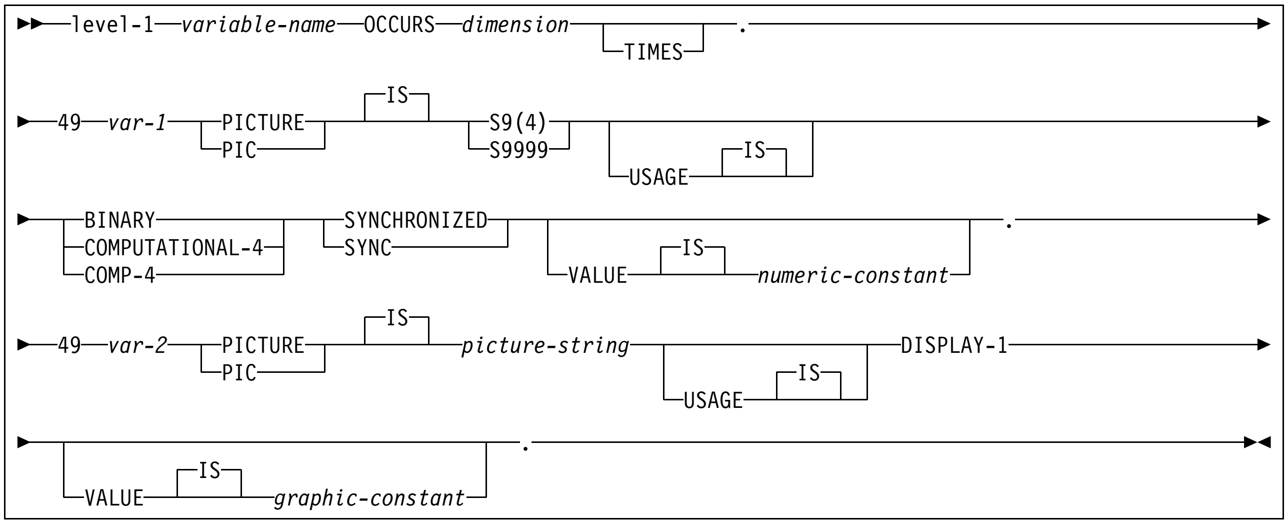


Figure 14. Varying length graphic

LOBs and LOB locator host variables arrays:

```
►►──level-1──variable-name──┬─────────────────┬──SQL TYPE IS─────────────────────────►
                            │         ┌─IS─┐  │
                            └─USAGE───┴────┴──┘

►──┬──┬──┬─BINARY LARGE OBJECT─┬───────┬──(──length──┬─────┬──)──┬──OCCURS──dimension──┬───────┬──.────►◄
   │  │  └─BLOB────────────────┘       │             ├─K─┤      │                      └─TIMES─┘
   │  ├──┬─CHARACTER LARGE OBJECT─┬────┤             ├─M─┤      │
   │  │  ├─CHAR LARGE OBJECT──────┤    │             └─G─┘      │
   │  │  └─CLOB───────────────────┘    │                        │
   │  └─DBCLOB─────────────────────────┘                        │
   └──┬─BLOB LOCATOR───┬────────────────────────────────────────┘
      ├─CLOB LOCATOR───┤
      └─DBCLOB LOCATOR─┘
```

Figure 15. LOB and LOB Locator

ROWID host variables arrays:

```
►►──level-1──variable-name──┬─────────────────┬──SQL TYPE IS ROWID──OCCURS──dimension──┬───────┬──.──►◄
                            │         ┌─IS─┐  │                                         └─TIMES─┘
                            └─USAGE───┴────┴──┘
```

Examples

Example 1: Declare a CURSOR C1 and fetch 10 rows using a multi-row FETCH statement.

```
01  OUTPUT-VARS.

    05  NAME OCCURS 10 TIMES.

        49 NAME-LEN   PIC S9(4) COMP-4 SYNC.

        49 NAME-DATA  PIC X(40).

    05  SERIAL-NUMBER PIC S9(9) COMP-4 OCCURS 10 TIMES.


PROCEDURE DIVISION.


    EXEC SQL

  DECLARE C1 CURSOR WITH ROWSET POSITIONING FOR

        SELECT NAME, SERIAL#

          FROM CORPORATE.EMPLOYEE

    END-EXEC.


    EXEC SQL

      OPEN C1

    END-EXEC.
```

```
EXEC SQL
  FETCH FIRST ROWSET FROM C1 FOR 10 ROWS
        INTO :NAME, :SERIAL-NUMBER
END-EXEC.
```

PL/I

Using host-variable-arrays

In PL/I programs, you can define an array. host-variable-arrays can be used with the multiple row FETCH and INSERT statements.

Numeric host-variable-arrays

The following figure shows the syntax for valid numeric host-variable-array declarations.

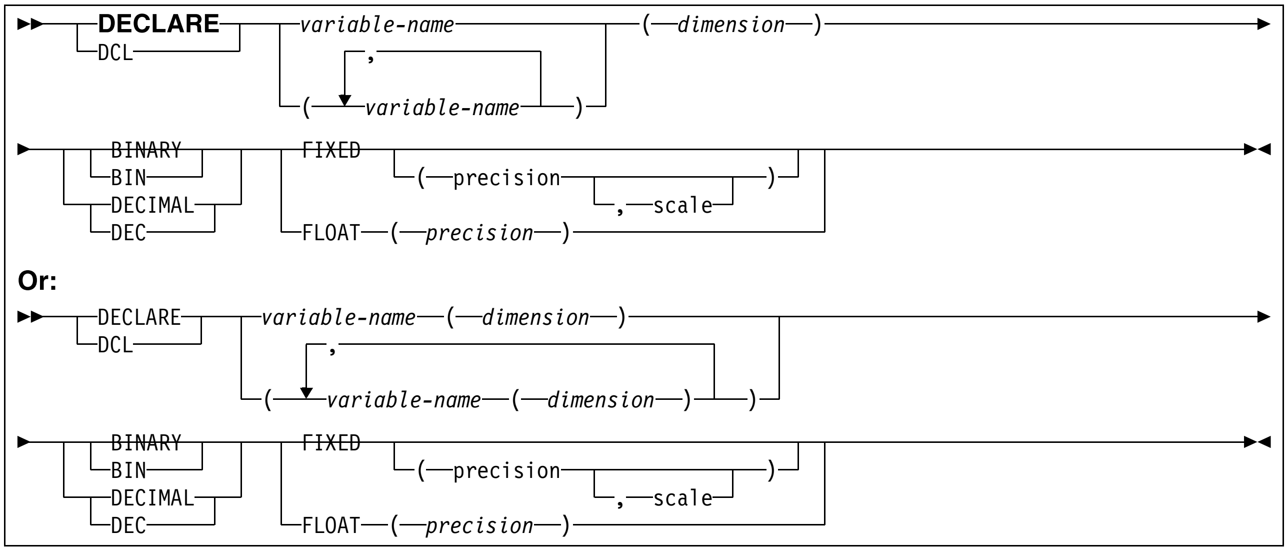


Figure 16. PL/I Numeric host-variable-arrays

Character arrays:

The following figure shows the syntax for valid character host-variable-array declarations.

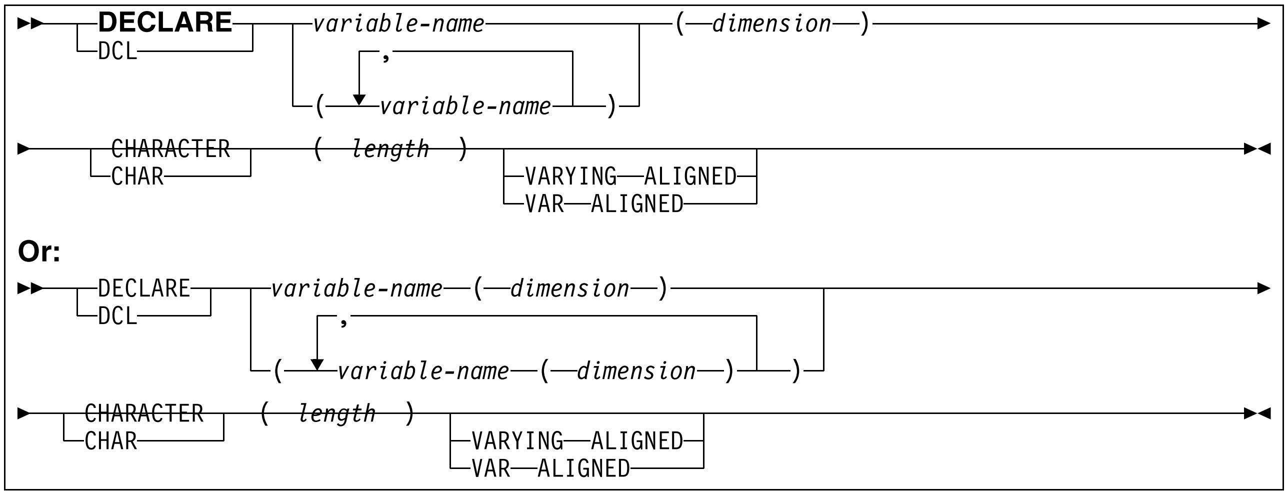


Figure 17. PL/I Character host-variable-arrays

The following figure shows the syntax for valid graphic host-variable-array declarations.

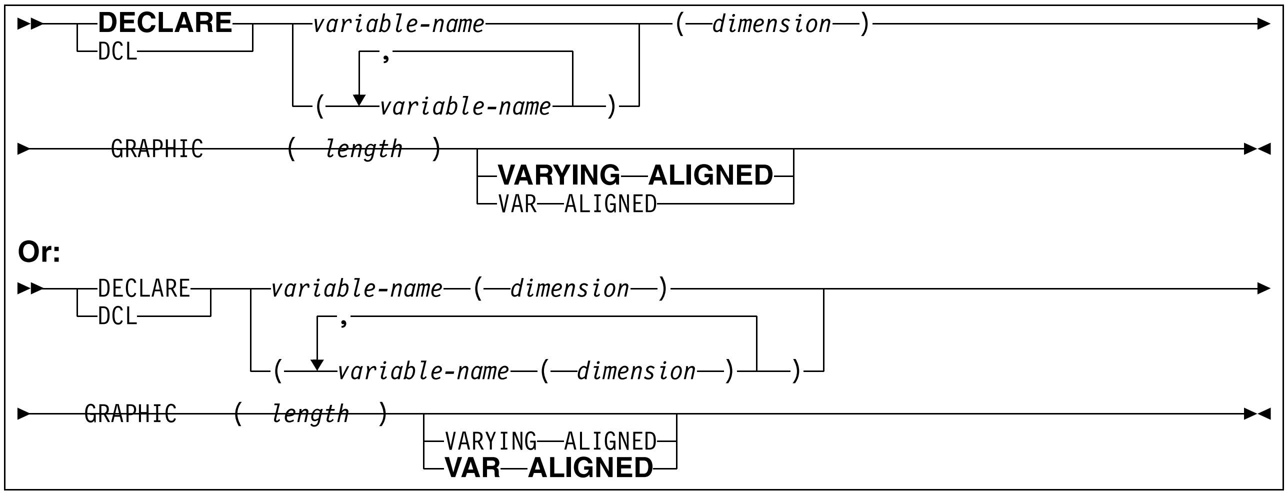


Figure 18. PL/I Graphic host-variable-arrays

LOBs and LOB locator host variables arrays:

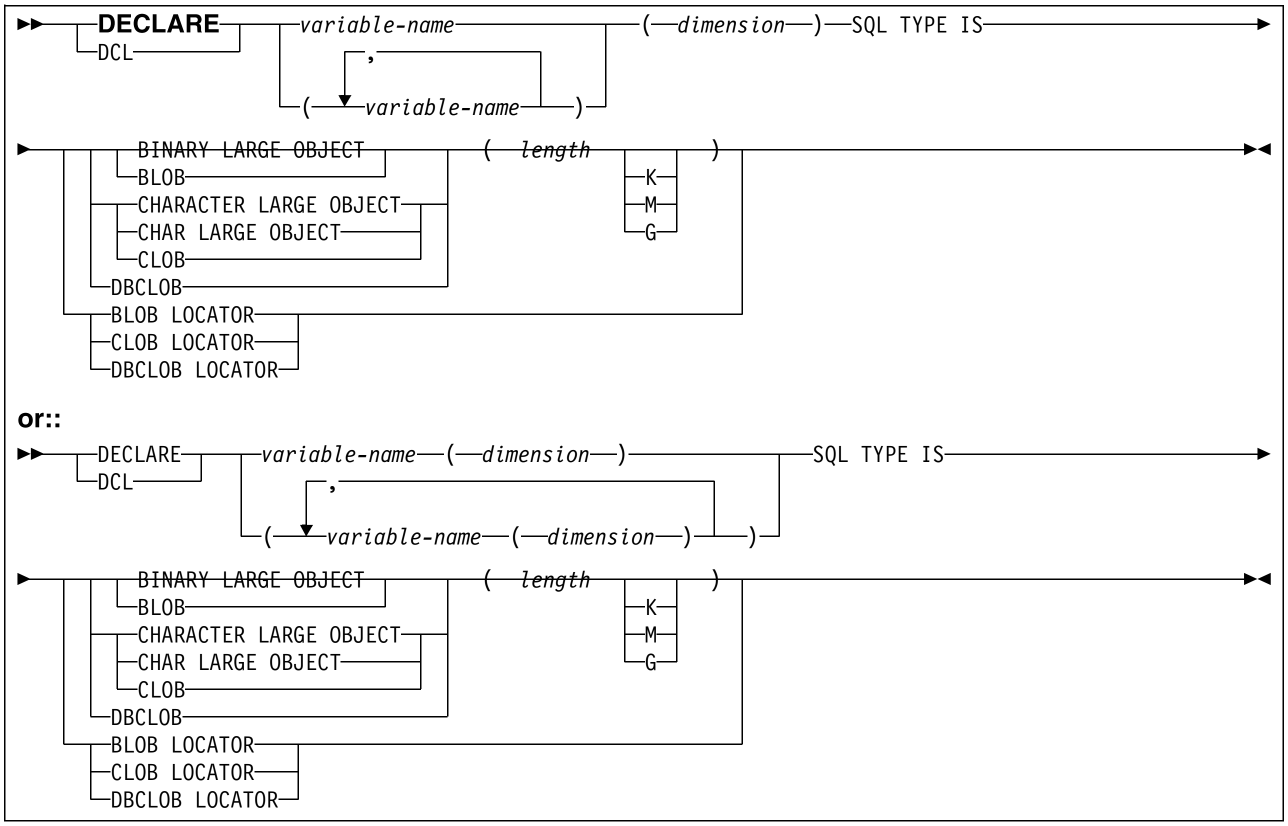


Figure 19. PL/I LOB host-variable-arrays

ROWID host variables arrays:

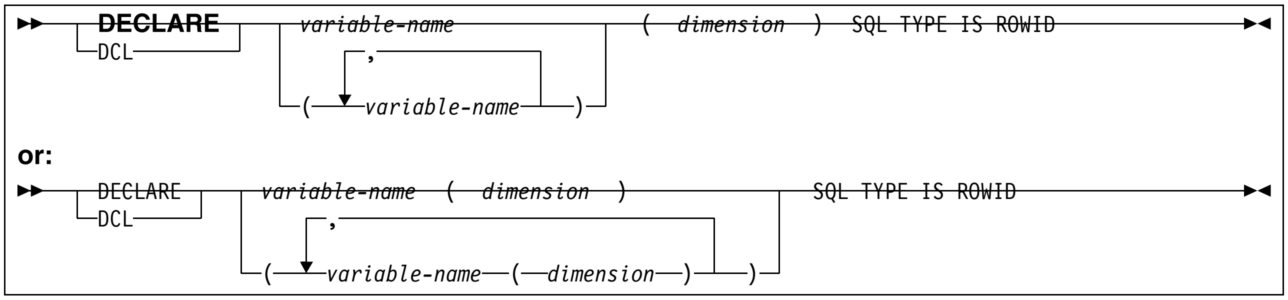


Figure 20. PL/I ROWID host-variable-arrays

Examples

Example 1:

You can retrieve 10 rows from the table CORPDATA.DEPARTMENT with:

```
DCL DEPTNO(10)    CHAR(3),
    DEPTNAME(10)  CHAR(29) VAR,
    MGRNO(10)     CHAR(6),
    ADMRDEPT(10)  CHAR(3);
```

```
DCL IND_ARRAY1(10) BIN FIXED(15);

DCL IND_ARRAY2(10) BIN FIXED(15);

DCL IND_ARRAY3(10) BIN FIXED(15);

DCL IND_ARRAY4(10) BIN FIXED(15);

   ....

EXEC SQL

   DECLARE C1 CURSOR WITH MULTIPLE ROW ACCESS FOR

      SELECT *

        FROM CORPDATA.DEPARTMENT;

   ...

EXEC SQL

   FETCH FIRST ROWSET FROM C1 FOR 10 ROWS INTO :DEPTNO :IND_ARRAY1,
                                              :DEPTNAME :IND_ARRAY2,
                                              :MGRNO :IND_ARRAY3,
                                              :ADMRDEPT :IND_ARRAY4;
```

ASSEMBLER

Assembler support for multi-row FETCH and INSERT is limited to the cases where USING DESCRIPTOR is allowed. The pre-compiler for Assembler does not recognize declarations of host-variable-arrays and will not create the proper descriptor (SQLDA) that is needed for multi-row statements. In the cases where USING DESCRIPTOR applies, the application programmer is responsible for allocating the storage correctly and for setting up the DESCRIPTOR according to the rules for USING DESCRIPTOR . See “USING DESCRIPTOR descriptor” on page 33 for INSERT and "Using DESCRIPTOR descriptor" on page 16 for FETCH

Considerations for using LOB host variables in all languages

LOB host variables that are referenced by the precompiler in SQL statements must be declared using the provisions provided by DB2 to declare LOB host variables (i.e. SQL TYPE IS ... as appropriate for the host language), or in a manner that is compatible to what the precompiler generates (that is to say, a LOB host variable contains a 31 bit length, followed by the data).

LOB host variables that are only referenced by an SQL statement that uses a DESCRIPTOR may have an alternate form. In this form, sqldatalen (or SQLDATAL, or SQLDATALEN depending on the language) contains a pointer to the length of the LOB, and sqldata (or SQLDATA depending on the language) contains a pointer to the data. For multi-row INSERT or FETCH statements, if this form of declaration for a LOB is used, sqldatalen should point to an array of integers, and sqldata will point at the first instance of the LOB data. Each LOB is a maximum of len.sqllonglen (or SQLLONGL or SQLLONGLEN depending on language) long. That is to say that the data for the first LOB should start at sqldata, the data for the second LOB should start at sqldata + len.sqllonglen +1 and so on. For INSERT statements, sqldatalen should could contain the actual length of the LOB. For FETCH statements, sqldatalen will contain the actual length of the LOB. Please note that when using this form of LOB declaration, LOB data will not be aligned on any particular boundary.

LOB host variables that are only referenced by an SQL statement that uses a DESCRIPTOR may have an alternate form. In this form, sqldatalen (or SQLDATAL, or SQLDATALEN depending on the language) contains a pointer to the length of the LOB, and sqldata (or SQLDATA depending on the language) contains a pointer to the data. For multi-row INSERT or FETCH statements, if this form of declaration for a LOB is used, sqldatalen should point to an array of integers, and sqldata will point at the first instance of the LOB data. Each LOB is a maximum of len.sqllonglen (or SQLLONGL or SQLLONGLEN depending on language) long. Although a LOB can be at most len.sqllonglen, the second LOB will start right after the first LOB in the area pointed to by sqldata. That is to say that the data for the first LOB should start at sqldata, the data for the second LOB should start at sqldata + sqldatalen(1) +1, the data for the third LOB will start at sqldata + sqldatalen(1) + sqldatalen(2) +2 and so on.

The advantage of this form is that memory will not be wasted for empty space. The disadvantage of this form is that applications will have to do more processing to calculate the location of the nth LOB in the memory pointed by sqldata. For INSERT statements, sqldatalen should could contain the actual length of the LOB. For FETCH statements, sqldatalen will contain the actual length of the LOB. Please note that when using this form of LOB declaration, LOB data will not be aligned on any particular boundary.

SQLVAR entries

Table 4. Fields in an occurrence of a base SQLVAR

C name assembler COBOL, or PL/I name	Data type	Usage in DESCRIBE[1] and PREPARE INTO	Usage in FETCH, OPEN, EXECUTE, and CALL
sqlname SQLNAME	VARCHAR(30)	Contains the unqualified name or label of the column, or a string of length zero if the name or label does not exist. If the prepared statement includes a UNION or UNION ALL clause, SQLNAME contains the name or label, if any, of the corresponding column of the first operand of the UNION. For DESCRIBE PROCEDURE, SQLNAME contains the cursor name used by the stored procedure to return the result set. The values for SQLNAME appear in the order the cursors were opened by the stored procedure. For DESCRIBE INPUT, SQLNAME is not used.	Can contain CCSID and/or host-variable-array dimension information. DB2 interprets the third and fourth byte of the data portion of SQLNAME as the CCSID of the host variable if all of the following are true: • The 6th byte of SQLDAID is '+' (x'4E') • SQLTYPE indicates the host variable is a string variable • The length of SQLNAME is 8 • The first two bytes of the data portion of SQLNAME are `X'0000'`. For **FETCH**, **OPEN, INSERT**, and **EXECUTE**, DB2 interprets the fifth through eighth bytes of the data portion of SQLNAME as a binary integer that represents the dimension of the host-variable-array, and corresponding indicator-array if one is specified, if all of the following are true: • The length of SQLNAME is 8 • The first two bytes of the data portion of SQLNAME are `X'0000'`.

Notes:

1. The third column of this table represents several forms of the DESCRIBE statement.
   - For DESCRIBE *output* and PREPARE INTO, the column pertains to columns of the result table.
   - For DESCRIBE CURSOR, the column pertains to a result set associated with a cursor.
   - For DESCRIBE INPUT, the column pertains to parameter markers.
   - For DESCRIBE PROCEDURE, the column pertains to the result sets returned by the stored procedure.

SQLCA Description of Fields

Table 5. Fields of SQLCA

assembler, COBOL, or PL/I Name	C Name	Fortran Name	Data type	Purpose
SQLERRD(3)	sqlerrd[2]	SQLERR(3)	INTEGER	Contains the number of rows affected after INSERT, UPDATE, and DELETE (but not rows deleted as a result of CASCADE delete). Set to 0 if the SQL statement fails, indicating that all changes made in executing the statement were canceled. Set to -1 for a mass delete from a table in a segmented table space. For rowset oriented fetch statements, contains the number of rows returned in the rowset. For SQLCODES -911 and -913, SQLERRD(3) contains the reason code for the timeout or deadlock .

Distributed Processing

Distributed Processing: Multi-row input and output processing requires DRDA SQLAM level 7 (see Open Group Technical Standard, DRDA Version 3).

Multi-row input and output processing and the GET DIAGNOSTICS statement are not supported in DB2 private protocol (SQLSTATE 56023, SQLCODE -512).

DB2 Commands

There are no Command changes for this line item

Utilities

There are no Utility changes for this line item

Chapter 3. Impact on User Tasks

The following paragraphs summarize the effects of this line item on the major tasks involved in using DB2.

Evaluating the Product

See “DECLARE CURSOR Statement” on page 6, “OPEN CURSOR” on page 7, “ALLOCATE CURSOR” on page 7, “FETCH Statement” on page 8, “PREPARE Statement” on page 24, “INSERT Statement” on page 25, “EXECUTE Statement” on page 32, “GET DIAGNOSTICS” on page 34, “Positioned Update” on page 46, “Positioned Delete” on page 47 and“Application Programming - Host Language Declarations” on page 49 for further information regarding functionality provided by this line item.

Planning for and Administering the Product

This category includes the following user tasks:

System Planning and Installing

N/A

Communicating with Other Systems

N/A

Database Design and Implementation

N/A

Security and Auditing

N/A

Operation and Recovery

N/A

Performance Monitoring and Tuning

Multi-row inserts as well as updates and deletes where current of cursor have the potential of expanding the unit of work within an application without having intermittent commit processes. This can affect the concurrency of users accessing a table space. Factors to tune include the size of the array and the use of commits between inserts, updates and deletes. The type of locking may also have a significant impact on concurrency and performance if lock escalation occurs.

Application Programming

See details regarding “DECLARE CURSOR Statement” on page 6, “FETCH Statement” on page 8, “PREPARE Statement” on page 24, “INSERT Statement” on page 25, “EXECUTE Statement” on page 32, “GET DIAGNOSTICS” on page 34, “Positioned Update” on page 46, “Positioned Delete” on page 47, and “Application Programming - Host Language Declarations” on page 49.

Chapter 4. User Task Guidance Information

In this section we describe

- "Planning for multi-row INSERT statements"
- "Planning for multi-row cursors and multi-row FETCH statements" on page 66

Planning for multi-row INSERT statements

Multi-row insert statements require no special planning.

Example of a multi-row INSERT

The following example demonstrates how a multi-row insert might be used:

Create a table

```
CREATE TABLE MY_EMP (ID INTEGER,

                     NAME VARCHAR(18));
```

Declare host variables, host variable arrays, and indicator arrays (assuming a C application)

```
long  hv1[10];

struct {

        short len;

        char data[18]} hv2[10];

short ind1[10], ind2[10], num_rows;
```

Initialize host variables

```
num_rows = 7;

for (i=0; i<=10; i++)

  {

  hv1[i] = i;

  ind1[i] = 0;

  imd2[i] = 0;

strcpy(hv2[i].data,"");

  }

strcpy(hv2[1].data,"Chris");

strcpy(hv2[3].data,"Patrick");

strcpy(hv2[5].data,"Terry");

strcpy(hv2[6].data,"Meg");
strcpy(hv2[8].data,"Maureen");

strcpy(hv2[9].data,"Helmut");

strcpy(hv2[10].data,"Peggy");


for (i=0; i<=10; i++)

  {

  hv2[i].len = strlen(hv2[i].data);

  }
```

Insert data into table

```
EXEC SQL INSERT INTO MY_EMP FOR :num_rows ROWS VALUES (:hv1:ind1,:hv2:ind2)
ATOMIC;
```

Seven rows of data will be inserted into table MY_EMP.

Planning for multi-row cursors and multi-row FETCH statements

Multi-row cursor may be used with scrollable cursors. Static scrollable cursors require that a TEMP database be allocated. No other special planning is required.

Example of a multi-row cursor and FETCH

The following example demonstrates how a multi-row fetch might be used (this example assumes the above table MY_EMP and seven rows of data)[17]:

Declare host variables, host variable arrays, and indicator arrays (assuming a C application)

```
long  hv1[10];

struct {

        short len;

        char data[18]} hv2[10];

short ind1[10], ind2[10], num_rows;
```

Initialize host variables

```
num_rows = 20;

for (i=0; i<=10; i++)

  {

  ind1[i] = 0;

  ind2[i] = 0;

strcpy(hv2[i].data,"");
```

17.Please note that these examples are intended to demonstrate examples of how the functionality in this line item could be used. They are not intended to endorse programming styles or methods.

```
  }
```

Now declare cursor

```
EXEC SQL DECLARE CS1 CURSOR WITH ROWSET POSITIONING FOR SELECT * FROM
MY_EMP;

EXEC SQL OPEN CS1;

EXEC SQL FETCH CS1 FOR :num_rows ROWS INTO :hv1:ind1, :hv2:ind2;
```

The 7 rows are returned in array entries 1-7. SQLCODE + 100 will be issued and SQLERRD3 will contain 7 (the number of rows returned). If there had been more than 10 rows in the table, SQLCODE -246 would have been returned after the fetch for the 10th row because there was no room in the host-variable-arrays to return the data.

Another example of how to fetch rows using a multi-row cursor follows (again, assuming the above table MY_EMP, data, and host language definitions):

Initialize host variables

```
num_rows = 10;

for (i=0; i<=10; i++)

  {

  ind1[i] = 0;

  imd2[i] = 0;

strcpy(hv2[i].data,"");

  }
```

Now declare cursor

```
EXEC SQL DECLARE CS1 CURSOR WITH ROWSET POSITIONING FOR SELECT * FROM
MY_EMP;
```

OPEN CURSOR

```
EXEC SQL OPEN CS1;
```

Fetch Ten Rows

```
EXEC SQL FETCH CS1 ROWSET STARTING AT ABSOLUTE 1 FOR :num_rows ROWS INTO
:hv1:ind1, :hv2:ind2;
```

The above application segment would fetch the 7 rows from MY_EMP. The first seven array entries would be filled with data. The SQLCODE would be +100, and SQLERRD3 would contain 7 (Seven rows returned). The array elements 1-7 would contain the data that was fetched.

Chapter 5. Instrumentation

There are no changes to instrumentation for this line item.

Chapter 6. Other Interfaces

Catalog and Directory

N/A

User-Maintained Tables or Databases

N/A

Installation

N/A

Log Records

N/A

Chapter 7. Installation, Migration, and Fallback

Installation

The install process is unchanged from the prior release.

There are no installation changes for this line item.

Migration

The migration process is unchanged from the prior release.

There are no changes to the migration process for this line item.

Incompatibilities after Migration

There are no incompatibilities after a migration.

There are no incompatibilities after a migration for this Line Item.

New/Modified SQL Reserved Words

VARIABLE will be added as a new SQL reserved word.

Fallback

The fall back process is unchanged from the prior release.

Compatibility Mode

The features of this line item are not available in Compatibility mode.

Enabling New Function Mode

The features of this line item are not available in Enabling New Function Mode.

New Function Mode

The features of this line item first become available in New Function Mode.

Line Item Considerations

n/a.

Incompatibilities after Fallback

There are no incompatibilities in the fallback release.

There are no incompatibilities after a fallback for this Line Item.

Coexistence

Coexistence does not exist in this line item because this function is not available until we are in New Function Mode.

Chapter 8. Messages and Codes

New SQL Codes

-227 **FETCH** *fetch-orientation* **IS NOT ALLOWED, BECAUSE CURSOR** *cursor-name* **HAS AN UNKNOWN POSITION (<sqlcode>,<sqlstate>)**

Explanation: The cursor position for *cursor-name* is unknown. A previous multiple-row-fetch for cursor *cursor-name* resulted in an error (SQLCODE *sqlcode*, SQLSTATE *sqlstate*) in the middle of processing multiple rows retrieved from DB2. One or more of the requested rows could not be returned to the program following the error, leaving the position of the cursor unknown.

If an indicator structure had been provided on the previous multiple row FETCH, a positive SQLCODE would have been returned and all of the rows retrieved from DB2 could have been returned to the application program.

System Action: The statement cannot be processed. The cursor position is not changed.

Programmer Response: Close and reopen the cursor to reset the position. For scrollable cursors, you can change the FETCH statement to specificy one of the other fetch orientations such as FIRST ROWSET, LAST ROWSET, ROWSET STARTING AT RELATIVE, or ROWSET STARTING AT ABSOLUTE to establish a valid rowset cursor position, and fetch multiple rows of data.

Destination: 24513

-246 **STATEMENT USING CURSOR** *cursor-name* **SPECIFIED NUMBER OF ROWS** *num-rows* **WHICH IS NOT VALID with** *dimension*

Explanation: A multiple-row FETCH or multiple-row INSERT statement is not valid. The number of rows specified is not greater than 0 and not less than or equal to 32767, or is greater than the dimension of one of the host-variable-arrays or indicator arrays. The number of rows specified is *num-rows* , and the dimension of the array is *dimension* . If this is a FETCH statement, the cursor name is *cursor-name* . Otherwise, the cursor name is not applicable.

System Action: The statement cannot be processed. The cursor position is not changed.

Programmer Response: Change the application to either declare, or allocate a host-variable-array that is large enough to contain the number of rows specified in the statement, or update the value of *num-rows*to a value within the valid range.

Destination: 42873

-230 **THE NUMBER OF ROWS FOR A ROWSET FOR CURSOR** *cursor-name* **HAS NOT BEEN SPECIFIED**

Explanation: A rowset positioned FETCH statement with a multiple-row-fetch clause that includes a FOR n ROWS clause must precede a rowset positioned FETCH statement that does not have a FOR n ROWS clause in order to establish the number of rows for a rowset cursor.

System Action: The statement cannot be executed.

Programmer Response: Change the FETCH statement to specify a FOR n ROWS clause, or change the order of the processing of the application to ensure that a FETCH statement with a FOR n ROWS clause is executed before the current FETCH statement is executed.

Destination: 24523

-247 **A HOLE WAS DETECTED ON A MULTIPLE ROW FETCH STATEMENT USING CURSOR** *cursor-name*, **BUT INDICATOR VARIABLES WERE NOT PROVIDED TO DETECT THE CONDITION**

Explanation: A hole was detected on a FETCH statement for multiple rows of data, but no indicator variables were provided to reflect the situation to the application.

System Action: The statement cannot be processed.

Programmer Response: Change the FETCH statement to provide at least one indicator variable and resubmit the statement.

Destination: 24519

-248 **A POSITIONED DELETE OR UPDATE STATEMENT FOR CURSOR** *cursor-name* **SPECIFIED ROW** *n* **OF A ROWSET, BUT THE ROW IS NOT CONTAINED WITHIN THE CURRENT ROWSET**

Explanation: The FOR ROW n OF ROWSET clause was specified on a positioned DELETE or UPDATE statement, but row *n* is not contained within the bound of the rowset.

System Action: The statement cannot be processed.

Programmer Response: Reissue the positioned UPDATE or DELETE with a value that is within the bounds of the rowset, or expand the bounds of the rowset by specifying that a larger number of rows be contained in the rowset using the FOR n ROWS clause on FETCH.

Destination: 24521

-249 **DEFINITION OF ROWSET ACCESS FOR CURSOR** *cursor-name* **IS INCONSISTENT WITH THE FETCH ORIENTATION CLAUSE** *clause* **SPECIFIED**

Explanation: The *clause* specified as the fetch-orientation for a FETCH statement is inconsistent with the definition of the cursor. What can be specified for fetch-orientation depends on whether the cursor was defined for rowset access:

- A cursor defined WITH ROWSET POSITIONING can only use rowset positioned fetch orientation clauses: NEXT ROWSET, PRIOR ROWSET, FIRST ROWSET, LAST ROWSET, CURRENT ROWSET, or ROWSET STARTING AT.
- A cursor defined WITHOUT ROWSET POSITIONING can only use row positioned fetch orientation keywords: NEXT, PRIOR, FIRST, LAST, BEFORE, AFTER, CURRENT, or ABSOLUTE, RELATIVE.

System Action: The statement cannot be processed.

Programmer Response: Correct the fetch orientation, or redefine the cursor.

Destination: 24523

-589 **A POSITIONED DELETE OR UPDATE STATEMENT FOR CURSOR** *cursor-name* **SPECIFIED A ROW OF A ROWSET, BUT THE CURSOR IS NOT POSITIONED ON A ROWSET**

Explanation: The FOR ROW n OF ROWSET clause was specified on a positioned DELETE or UPDATE statement, but the cursor is not currently positioned on a rowset.

System Action: The statement cannot be processed.

Programmer Response: Issue a FETCH statement to position the cursor on the desired rowset, and then reissue the positioned DELETE or UPDATE statement. If the cursor is not defined for rowset access, redefine the cursor first.

Destination: 24520

-20185 **CURSOR** *cursor-name* **IS NOT DEFINED TO ACCESS ROWSETS, BUT A CLAUSE WAS SPECIFIED THAT IS VALID ONLY WITH ROWSET ACCESS**

Explanation: The FOR n ROWS clause was specified on a FETCH statement, but the cursor is not defined for rowset access.

System Action: The statement cannot be processed.

Programmer Response: Remove the FOR n ROWS clause from the FETCH statement, or redefine the cursor for multiple row access with the WITH ROWSET POSITIONING clause on DECLARE CURSOR or PREPARE.

Destination: 24518

-20186 **A CLAUSE WAS SPECIFIED THAT IS NOT VALID FOR THE STATEMENT BEING PREPARED OR EXECUTED**

Explanation: A clause was not valid for one of the following reasons:

- On a PREPARE statement, a FOR SINGLE ROW or FOR MULTIPLE ROWS clause was specified, but the statement being prepared is not an INSERT statement.
- On a PREPARE statement, an ATOMIC or NOT ATOMIC clause was specified, but the statement being prepared is not an INSERT statement.
- On an EXECUTE statement, a multiple-row-insert clause was specified but the statement being executed is not an INSERT statement.

System Action: The statement cannot be processed.

Programmer Response: Correct the statement by either removing the WITH or WITHOUT VARYING ROW COUNT clause, ATOMIC or NOT ATOMIC clause, or ensure the statement being executed is an INSERT statement.

Destination: 07501

-30106 **INVALID INPUT DATA DETECTED FOR A MULTIPLE ROW INSERT OPERATION. INSERT PROCESSING IS TERMINATED**

Explanation: An error was detected with the input data for one row of a multiple row INSERT operation. No further rows will be inserted. For an atomic operation, all inserted rows are rolled back. For a non-atomic operation, rows inserted successfully before the row containing the invalid input data was encountered are not rolled back.

System Action: The statement cannot be processed.

Programmer Response: Correct the row containing the invalid input data and submit the multiple-row INSERT statement again for the rows that did not get inserted.

Destination: 22527

-393 **THE CONDITION OR CONNECTION NUMBER IS INVALID**

Explanation: The value of the CONDITION or CONNECTION number specified in the GET DIAGNOSTICS statement is either less than zero, or greater than the number of available diagnostics.

System Action: The statement cannot be processed.

Programmer Response: Correct the value of the CONDITION or CONNECTION number, ensuring the number is between 1 and the value of the NUMBER statement-information item or GET DIAGNOSTICS. Resubmit the GET DIAGNOSTICS CONDITION or GET DIAGNOSTICS CONNECTION statement.

Destination: 35000

Revised SQLCODES

-5012 **HOST VARIABLE** *host-variable* **IS NOT AN EXACT NUMERIC WITH SCALE ZERO**

Explanation: A host variable *host-variable* was specified, but it is not valid in the context in which it was used. Host variable *host-variable* was specified as part of ABSOLUTE or RELATIVE in a FETCH statement, or in a FOR n ROWS clause of a FETCH or INSERT statement. The host variable was not usable for one of the following reasons:

- It is not an exact numeric type
- The scale is not zero

System Action: The statement cannot be processed.

Programmer Response: Change the host variable to an exact numeric type with a scale of zero.

Destination: 42618

-225 **FETCH STATEMENT FOR CURSOR** *cursor-name* **IS NOT VALID BECAUSE THE CURSOR IS NOT DEFINED AS SCROLL**

Explanation: A FETCH statement for cursor *cursor-name* has been specified and one of the following errors has occurred:

- PRIOR, FIRST, LAST, BEFORE, AFTER, CURRENT, ABSOLUTE or RELATIVE was specified as a row-positioned fetch orientation on FETCH, but the cursor is not defined as a scrollable cursor. NEXT is the only row-positioned fetch orientation that can be specified for cursors that are not scrollable.
- PRIOR ROWSET, FIRST ROWSET, LAST ROWSET, CURRENT ROWSET, or ROWSET STARTING AT was specified as a row-positioned fetch orientation on FETCH, but the cursor is not defined as a scrollable cursor. NEXT ROWSET is the only rowset-positioned fetch orientation that can be specified for cursors that are not scrollable.

System Action: The statement cannot be processed. The cursor position is unchanged.

Programmer Response: Change the FETCH statement to remove the fetch orientation clause (i.e., PRIOR, FIRST, PRIOR ROWSET, FIRST ROWSET etc.), and change it to NEXT or NEXT ROWSET. Alternatively, you could change the definition of the cursor to be scrollable.

Destination: 42872

-301 **THE VALUE OF INPUT HOST VARIABLE OR PARAMETER NUMBER** *position-number* **CANNOT BE USED AS SPECIFIED BECAUSE OF ITS DATA TYPE**

Explanation: DB2 received data that could not be used as specified in the statement because its data type is incompatible with the requested operation.

The *position-number* identifies either the host variable number (if the message is issued as a result of an INSERT, UPDATE, DELETE, SELECT, VALUE INTO, or SET assignment statement), GET DIAGNOSTICS statement, or the parameter number (if the message is issued as the result of a CALL statement, or the invocation of a function).

System Action: The statement cannot be executed.

Programmer Response: Correct the application program, function or stored procedure. Ensure that the data type of the indicated input host variable or parameter in the statement is compatible with the way it is used.

Destination: 42895

New Messages

DSNH5011I **HOST VARIABLE ARRAY** *host-variable-array* **IS EITHER NOT DEFINED OR IS NOT USABLE**

Explanation: Host-variable-array *host-variable-array* was specified in a multiple-row FETCH, or multiple-row INSERT statement. The host-variable-array is not defined or is not usable for one of the following reasons:

- The host variable is not a valid host-variable-array.
- The host variable is not a dimensioned array.
- The host-variable-array has more than 1 dimension.
- The host-variable-array defines a structure that does not conform to the rules for defining a host-variable-array.
- The dimension of the host-variable-array used for indicators is not equal to the dimension of the main host-variable-array.

System Action: The statement cannot be processed.

Programmer Response: Correct any of the following and precompile the program again. Ensure that

- the dimension of the indicator variable array is equal to the dimension of the main host-variable-array.
- the host-variable-array is a single-dimensional array.
- the host-variable-array or host indicator variable array do not contain structures that are not permissible. The only structures that may be defined in a host-variable-array are those that are

used to define variable length string host variables.

See the DB2 for z/OS Application Programming and SQL Guide for more language specific information on how to define the host-variable-arrays in a program.

Chapter 9. Dependencies

Function-Dependent Hardware Requirements

None.

Function-Dependent Hardware Requirements

n/a.

Function-Dependent Program Requirements

Remote clients or servers will be required to support The Open Group DRDA V3 standard. Additional IBM compliant products: DB2 Connect V8, DB2 V8 Server for Unix, Windows, and Linux, and JDBC Type 4 Driver.

Chapter 10. Performance

Performance Objectives

This line item is primarily a functional improvement. There will be changes to the performance of applications that utilize the function provided by this line item. The changes are detailed below.

Expected Improvements/Degradations

The following items list expected performance changes:

1. Using a multi-row fetch read should take less time than a cursor open with multiple fetches.
2. Using a multi-row insert should take less time than multiple inserts.

The above statements apply to local applications. Remote applications should see better improvement because API crossing is more expensive than it is for local applications.

Remote applications with updateable cursors should also see an improvement in performance of fetching on these cursors. The functional changes introduced with this line item will allow blocking on cursors where no blocking was allowed before.

Performance Evaluation

Performance evaluation will include:

- Comparison of performance improvement/degradation from current method of fetching or inserting multiple rows to this new method.
- Evaluation of impact to network traffic in a client/server architecture due to a lack of a blocking factor.
- Impact of multi-row insert and update/delete where current of on concurrency.
- Comparison of cursor open performance in Version 7 vs. cursor open performance in Version 8.

Factors Affecting Performance

The number of rows being inserted into a table, using a multi-row insert statement, might affect concurrency.

Chapter 11. Standards

The functions described in this specification conform to The Open Group Version 3 DRDA Standard.

Chapter 12. National Language Support Considerations

n/a

Chapter 13. Implementation Notes

This line item is attempting to meet and resolve the following (sometimes conflicting) requirements:

1. Our ERP vendors have significant influence on not only what function is implemented in DB2 but also, how that function is implemented in our product. Since dynamic processing is key to how well these tools perform, the performance requirements are geared towards making dynamic processing fast and easy to use. These requirements are also beneficial to our customer base as well. The concerns are:
   - Dynamic Caching: Vendors require that dynamic statements make as much use of this feature of DB2 for OS/390 as possible. If the value of "n" rows changes in a statement or the variables change, it is important that the statement still qualify for dynamic caching. This reduces the number of statements (PREPAREs) that an application has to execute, making application execution time much faster.
   - Continue on Error Processing: If a statement is a blocked INSERT and, for example, 1 row out of a total of 100 multiple-insert rows fails (i.e., due to a duplicate row error), it is costly for the vendor tool to continually re-work and re-submit the SQL statement, until all potential duplicates or problems are found one at a time. Optimally, the multiple-row insert should continue through errors, and process/insert all the rows that it can.
2. Compatibility with the workstation is another issue that requires consideration within the context of this line item. Unfortunately, we will be implementing forms of the multiple-row FETCH/INSERT which are not implemented in the AS/400 or UWO. (For documentation purposes, the AS/400 has implemented a host structure array and row storage form for multiple-row INSERT. UWO has implemented a host variable (not array) form of multiple-row INSERT. UWO has not yet implemented the multiple-row FETCH statement.)
3. Locking: There are two very conflicting reasons for using multiple-row fetch. For "window" scrolling, a user will want to view a specific set of rows of data, and at the same time will want to have the ability to change any of the viewed rows. This implies that the set of rows are "locked" and not able to be changed while this user holds the n-row set of data. The other type of user that will want to use multiple-row fetch, is the user that is very performance oriented. This type of application wants to be able to fetch (both remotely as well as locally) a set number of rows with only one "trip" into DB2. For this approach, locks should NOT be held on the set of rows, as no changes to the data are intended. Both of the these methods have merit (i.e., customer interest). - It was decided, after the DRB tutorial, to remove the KEEP LOCK clause pending further feedback from the ODBC team. The lock portion of SQL151 may be DCRed in later.
4. SLC evolution: The formats of the multiple-row SQL statements are not yet set by the SQL Language Council, and decisions need to be made and agreed to about the syntax and semantics of these statements. Consequently, we must try to ensure that the formats we choose will not be incompatible with the final decisions of the SQL Language Council.
5. Because of resource constraints, and because the syntax of the multiple-row FETCH and INSERT statements are necessitating a change to host variable language declarations, we need to implement these changes in the most popularly used languages for our product. Consequently the declaration changes to implement the multiple-row statements, will be made for C, C++, COBOL, and PL/I.

6. This INTERNAL section will be removed from document. It is for documentation purposes only. Within our product there are certain requirements as well as limitations:
   - Don't change MSIB or internal too much
   - Storage considerations when moving in or storing multiple rows of information
   - Limit the number of types of statements
   - Possible limitation on the number of languages, especially if implementing host_struct arrays form.

For documentation purposes the allowable syntax for the multirow FETCH/INSERT statements are listed

below:

For discussion purposes, the various formats of the multiple-row statements are shown below.

There are three forms of the multiple-row FETCH statement:

• Fetch with SQLDA and storage area:

FETCH RELATIVE 4 FROM TABLE1 FOR 6 ROWS

USING DESCRIPTOR :*sqlda INTO :*row_area;

• Fetch with a host variable array:

FETCH ABSOLUTE 10

FROM CURS1

FOR 6 ROWS

INTO :hav1, :hva2, :hva3 ;

• Fetch with a host structure array:

FETCH FROM TABLE1 FOR :n ROWS INTO :host_struct_array;

There are 10 forms of multiple-row INSERT.

1. Static INSERT with a host variables (not arrays)

INSERT INTO T :hv1, :hv2;

2. Dynamic INSERT with a host variables (not arrays)

stmt = 'INSERT INTO T VALUES ?,?';

3. Static INSERT with host variables (not arrays):

INSERT INTO T VALUES (1,:a), (:b, :c);

4. Dynamic INSERT with host variables (not arrays):

stmt = 'INSERT INTO T VALUES (1,?), (?,?)';

5. Static INSERT with host variable arrays:

INSERT INTO T FOR :n ROWS

VALUES(:hva1, :hva2);

6. Dynamic INSERT with host variable arrays:

stmt = 'INSERT INTO T VALUES(?, ?)';

7. Static INSERT with host struct array:

INSERT INTO T FOR :n ROWS

VALUES(:host-struct-array);

8. Dynamic INSERT with host struct array:

stmt = 'INSERT INTO T VALUES(?);

9. Static INSERT with row storage format:

INSERT INTO T FOR :n ROWS USING DESCRIPTOR :sqldaptr

VALUES(:rowArea) NOT ATOMIC SET :diagnostic_array;

10. Dynamic INSERT with row storage format:

attr = 'FOR MULTIPLE ROWS';

stmt = 'INSERT INTO T1 USING DESCRIPTOR ?

VALUES(?)';

Toronto has implemented forms three and four of the INSERT examples above. Toronto has not implemented

multiple-row FETCH. The AS/400 has implemented forms nine and ten of the INSERT examples.

Chapter 14. Terminology

ATOMIC - Specifies that all of the rows should be treated as a single unit. If any row of the SQL statement fails, then all fail.

NOT ATOMIC - Specifies that all of the rows should be not be treated as a single unit. So, regardless of any failure of any particular row, no attempt will be made to undo other successful actions.

cursor A named control structure used by an application program to point to one or more specific rows within some ordered set of rows of the result table. The cursor is used to retrieve rows from the result table (with a FETCH statement), and possibly to make updates or deletes to corresponding rows in the database. A cursor is defined with a DECLARE CURSOR statement. A cursor can be defined to always return a single row, or to possibly return multiple rows depending on what is specified on the FETCH statement.

multiple row cursor A cursor defined such that multiple rows can be returned for a single FETCH statement. The FETCH statement indicates how many rows are to be retrieved.

rowset A set of rows that is retrieved through a multiple-row fetch.

host-variable-array An array in which each element of the array corresponds to a value for a column. The dimension of the array determines the maximum number of rows that the array can be used for.

Chapter 15. Sizing

The total estimated size at line item DR2 exit is xx.x KLOC without contingency and 9.5 KLOC with 50% contingency. The estimate is broken down by component in Table 6.

Table 6. Estimated Lines of Code without contingency. . Sizing broken down by components

Department	**Component**	**Resource**	**Description**	**LOC**
M55	DM	Karelle Cornwell, Ester Mote	Support for New ROWSET cursor concept.	1000
OF2	Parser/Precompiler	Marion Farber	New Syntax and Statements	2000
D3T	Runtime/Strgen	Daya Vivek, Helen Tjho, Andrei Lurie	STRGEN and Runtime support for New Statements	4000
D3T	EXEC	Georgia Fuller	Executives Support	500
L09	Dist	James Pickel, Margaret Dong, Wendy Koontz	Support for DRDA function and performance enhancements	2000
Totals				9500

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.