

# Using Natural Statements and System Variables

This section contains special considerations concerning Natural data manipulation language (DML) statements (that is, Natural native DML statements and Natural SQL DML statements), and Natural system variables when used with DB2.

It mainly consists of information also contained in the Natural basic documentation set where each Natural statement and variable is described in detail.

For an explanation of the symbols used in this section to describe the syntax of Natural statements, see *Syntax Symbols* in the *Natural Statements* documentation.

For information on logging SQL statements contained in a Natural program, refer to *DBLOG Trace Screen for SQL Statements* in the *DBLOG Utility* documentation.

This section covers the following topics:

- DB2 Special Register Consideration
  - Using Natural Native DML Statements
  - Using Natural SQL Statements
  - Using Natural System Variables
  - Multiple Row Processing
  - Error Handling
- 

## DB2 Special Register Consideration

NDB refreshes the following DB2 special registers automatically to the values, which applied to the least previous executed transaction.

- CURRENT SQLID
- CURRENT SCHEMA
- CURRENT PATH
- CURRENT PACKAGE PATH

NDB refreshes the following DB2 special registers only automatically to the values, which applied to the least previous executed transaction, if the parameter REFRESH=ON is set.

- CURRENT PACKAGESET

- CURRENT SERVER

Those special registers are refreshed regardless whether the previously executed transaction was rolled back or was committed.

All other special registers are not implicitly manipulated by NDB.

## Using Natural Native DML Statements

This section summarizes particular points you have to consider when using Natural data manipulation language (DML) statements with DB2. Any Natural statement not mentioned in this section can be used with DB2 without restriction.

Below is information on the following Natural DML statements:

- BACKOUT TRANSACTION
- DELETE
- END TRANSACTION
- FIND
- HISTOGRAM
- READ
- STORE
- UPDATE

### BACKOUT TRANSACTION

The Natural native DML statement `BACKOUT TRANSACTION` undoes all database modifications made since the beginning of the last logical transaction. Logical transactions can start either after the beginning of a session or after the last `SYNCPOINT`, `END TRANSACTION`, or `BACKOUT TRANSACTION` statement.

How the statement is translated and which command is actually issued depends on the TP-monitor environment:

- If this command is executed within a Natural stored procedure or Natural user-defined function (UDF), Natural for DB2 executes the underlying rollback operation. This sets the caller into a must-rollback state. If this command is executed within a Natural stored procedure or UDF for Natural error processing (implicit `ROLLBACK`), Natural for DB2 does not execute the underlying rollback operation, thus allowing the caller to receive the original Natural error.
- Under CICS, the `BACKOUT TRANSACTION` statement is translated into an `EXEC CICS ROLLBACK` command. However, in pseudo-conversational mode, only changes made to the database since the last terminal I/O are undone. This is due to CICS-specific transaction processing, see *Natural for DB2 under CICS*.

**Note:**

Be aware that with terminal input in database loops, Natural switches to conversational mode if no file server is used.

- In batch mode and under TSO, the BACKOUT TRANSACTION statement is translated into an SQL ROLLBACK command.

**Note:**

If running in a DSNMTV01 environment, the BACKOUT TRANSACTION statement is ignored if the used PSB has been generated without the CMPAT=YES option.

- Under IMS TM, the BACKOUT TRANSACTION statement is translated into an IMS Rollback (ROLB) command. However, only changes made to the database since the last terminal I/O are undone. This is due to IMS TM-specific transaction processing, see *Natural for DB2 under IMS TM*.

As all cursors are closed when a logical unit of work ends, a BACKOUT TRANSACTION statement must not be placed within a database loop; instead, it has to be placed outside such a loop or after the outermost loop of nested loops.

If an external program written in another standard programming language is called from a Natural program, this external program must not contain its own ROLLBACK command if the Natural program issues database calls, too. The calling Natural program must issue the BACKOUT TRANSACTION statement for the external program.

If a program tries to backout updates which have already been committed, for example by a terminal I/O, a corresponding Natural error message (NAT3711) is returned.

**DELETE**

The Natural native DML statement DELETE is used to delete a row from a table which has been read with a preceding FIND, READ, or SELECT statement. It corresponds to the SQL statement DELETE WHERE CURRENT OF *cursor-name*, which means that only the row which was read last can be deleted.

Example:

```
FIND EMPLOYEES WITH NAME = 'SMITH'
      AND FIRST_NAME = 'ROGER'
DELETE
```

Natural would translate the above Natural statements into SQL and assign a cursor name (for example, CURSOR1) as follows:

```
DECLARE CURSOR1 CURSOR FOR
SELECT FROM EMPLOYEES
      WHERE NAME = 'SMITH' AND FIRST_NAME = 'ROGER' FOR UPDATE OF NAME
DELETE FROM EMPLOYEES
      WHERE CURRENT OF CURSOR1
```

Both the SELECT and the DELETE statement refer to the same cursor.

Natural translates a Natural native DML DELETE statement into a Natural SQL DELETE statement in the same way it translates a Natural native DML FIND statement into a Natural SQL SELECT statement.

A row read with a `FIND SORTED BY` cannot be deleted due to DB2 restrictions explained with the `FIND` statement. A row read with a `READ LOGICAL` cannot be deleted either.

### **DELETE when Using the File Server**

If a row rolled out to the file server is to be deleted, Natural rereads automatically the original row from the database to compare it with its image stored in the file server. If the original row has not been modified in the meantime, the `DELETE` operation is performed. With the next terminal I/O, the transaction is terminated, and the row is deleted from the actual database.

If the `DELETE` operates on a scrollable cursor, the row on the file server is marked as `DELETE` hole and is deleted from the base table.

However, if any modification is detected, the row will not be deleted and Natural issues the NAT3703 error message for non-scrollable cursors.

If the `DELETE` operates on a scrollable cursor, Natural for DB2 simulates `SQLCODE -224 THE RESULT TABLE DOES NOT AGREE WITH THE BASE TABLE USING` for DB2 compliance.

If the `DELETE` operates on a scrollable cursor and the row has become a hole, Natural for DB2 simulates `SQLCODE -222 AN UPDATE OR DELETE OPERATION WAS ATTEMPTED AGAINST A HOLE`.

Since a `DELETE` statement requires that Natural rereads a single row, a unique index must be available for the respective table. All columns which comprise the unique index must be part of the corresponding Natural view.

### **END TRANSACTION**

The Natural native DML statement `END TRANSACTION` indicates the end of a logical transaction and releases all DB2 data locked during the transaction. All data modifications are committed and made permanent.

How the statement is translated and which command is actually issued depends on the TP-monitor environment:

- If this command is executed from a Natural stored procedure or user defined function (UDF), Natural for DB2 does not execute the underlying commit operation. This allows the stored procedure or UDF to commit updates against non DB2 databases.
- Under CICS, the `END TRANSACTION` statement is translated into an `EXEC CICS SYNCPOINT` command. If the file server is used, an implicit end-of-transaction is issued after each terminal I/O. This is due to CICS-specific transaction processing in pseudo-conversational mode, see *Natural for DB2 under CICS*.
- In batch mode and under TSO, the `END TRANSACTION` statement is translated into an `SQL COMMIT WORK` command.

#### **Note:**

If running in a DSNMTV01 environment the `END TRANSACTION` statement is ignored if the used PSB has been generated without the `CMPAT=YES` option.

- Under IMS TM, the END TRANSACTION statement is not translated into an IMS CHKP call, but is ignored. Due to IMS TM-specific transaction processing (see *Natural for DB2 under IMS TM*), an implicit end-of-transaction is issued after each terminal I/O.

Except when used in combination with the SQL WITH HOLD clause (see SELECT – SQL in *Using Natural SQL Statements*), an END TRANSACTION statement must not be placed within a database loop, since all cursors are closed when a logical unit of work ends. Instead, it has to be placed outside such a loop or after the outermost loop of nested loops.

If an external program written in another standard programming language is called from a Natural program, this external program must not contain its own COMMIT command if the Natural program issues database calls, too. The calling Natural program must issue the END TRANSACTION statement on behalf of the external program.

**Note:**

With DB2, the END TRANSACTION statement cannot be used to store transaction data.

## FIND

The Natural native DML statement FIND corresponds to the Natural SQL statement SELECT.

Example:

Natural native DML statements:

```
FIND EMPLOYEES WITH NAME = 'BLACKMORE'
      AND AGE EQ 20 THRU 40
OBTAIN PERSONNEL_ID NAME AGE
```

Equivalent Natural SQL statement:

```
SELECT PERSONNEL_ID, NAME, AGE
FROM EMPLOYEES
WHERE NAME = 'BLACKMORE'
      AND AGE BETWEEN 20 AND 40
```

Natural internally translates a FIND statement into an SQL SELECT statement as described in *Processing of SQL Statements Issued by Natural* in the section *Internal Handling of Dynamic Statements*. The SELECT statement is executed by an OPEN CURSOR statement followed by a FETCH command. The FETCH command is executed repeatedly until either all records have been read or the program flow exits the FIND processing loop. A CLOSE CURSOR command ends the SELECT processing.

The WITH clause of a FIND statement is converted to the WHERE clause of the SELECT statement. The basic search criterion for a DB2 table can be specified in the same way as for an Adabas file. This implies that only database fields which are defined as descriptors can be used to construct basic search criteria and that descriptors cannot be compared with other fields of the Natural view (that is, database fields) but only with program variables or constants.

**Note:**

As each database field (column) of a DB2 table can be used for searching, any database field can be defined as a descriptor in a Natural DDM.

The WHERE clause of the FIND statement is evaluated by Natural *after* the rows have been selected via the WITH clause. Within the WHERE clause, non-descriptors can be used and database fields can be compared with other database fields.

**Note:**

DB2 does not have sub-, super-, or phonetic descriptors.

A FIND NUMBER statement is translated into a SELECT statement containing a COUNT ( \* ) clause. The number of rows found is returned in the Natural system variable \*NUMBER as described in the Natural *System Variables* documentation.

The FIND UNIQUE statement can be used to ensure that only one record is selected for processing. If the FIND UNIQUE statement is referenced by an UPDATE statement, a non-cursor (Searched) UPDATE operation is generated instead of a cursor-oriented (Positioned) UPDATE operation. Therefore, it can be used if you want to update a DB2 primary key. It is, however, recommended to use the Natural SQL Searched UPDATE statement to update a primary key.

In static mode, the FIND NUMBER and FIND UNIQUE statements are translated into a SELECT SINGLE statement as described in the section *Using Natural SQL Statements*.

The FIND FIRST statement cannot be used. The PASSWORD, CIPHER, COUPLED and RETAIN clauses cannot be used either.

The SORTED BY clause of a FIND statement is translated into the SQL SELECT . . . ORDER BY clause, which follows the search criterion. Because this produces a read-only result table, a row read with a FIND statement that contains a SORTED BY clause cannot be updated or deleted.

A limit on the depth of nested database loops can be specified at installation time. If this limit is exceeded, a Natural error message is returned.

**Notes:**

1. If a processing limit is specified as a constant integer number, for example, FIND ( 5 ), the limitation value will be translated into a FETCH FIRST *integer* ROWS ONLY clause in the generated SQL string.
2. Natural for DB2 supports DB2 multiple row processing on behalf of the MULTIFETCH clause of the FIND statement.

**FIND when using the File Server**

As far as the file server is concerned, there are no programming restrictions with selection statements. It is, however, recommended to make yourself familiar with its functionality considering performance and file server space requirements.

**HISTOGRAM**

The Natural DML statement HISTOGRAM returns the number of rows in a table which have the same value in a specific column. The number of rows is returned in the Natural system variable \*NUMBER as described in the Natural *System Variables* documentation.

Example:

Natural native DML statements:

```
HISTOGRAM EMPLOYEES FOR AGE
OBTAIN AGE
```

Equivalent Natural SQL statement:

```
SELECT COUNT(*) , AGE FROM EMPLOYEES
WHERE AGE > -999
GROUP BY AGE
ORDER BY AGE
```

Natural translates the HISTOGRAM statement into an SQL SELECT statement, which means that the control flow is similar to the flow explained for the FIND statement.

**Note:**

With Universal Database Server for z/OS Version 8, Natural for DB2 supports DB2 multiple row processing on behalf of the MULTIFETCH clause of the HISTOGRAM statement.

## READ

The Natural native DML statement READ can also be used to access DB2 tables. Natural translates a READ statement into an SQL SELECT statement.

READ PHYSICAL and READ LOGICAL can be used; READ BY ISN, however, cannot be used, as there is no DB2 equivalent to Adabas ISNs. The PASSWORD and CIPHER clauses cannot be used either.

Since a READ LOGICAL statement is translated into a SELECT . . . ORDER BY statement, which produces a read-only table, a row read with a READ LOGICAL statement cannot be updated or deleted (see Example 1). The start value can only be a constant or a program variable; any other field of the Natural view (that is, any database field) cannot be used.

A READ PHYSICAL statement is translated into a SELECT statement without an ORDER BY clause and can therefore be updated or deleted (see Example 2).

Example 1:

Natural native DML statements:

```
READ PERSONNEL BY NAME
OBTAIN NAME FIRSTNAME DATEOFBIRTH
```

Equivalent Natural SQL statement:

```
SELECT NAME, FIRSTNAME, DATEOFBIRTH FROM PERSONNEL
WHERE NAME >= ' '
ORDER BY NAME
```

Example 2:

The Natural native DML statements:

```
READ PERSONNEL PHYSICAL
OBTAIN NAME
```

Equivalent Natural SQL statement:

```
SELECT NAME FROM PERSONNEL
```

If the READ statement contains a WHERE clause, this clause is evaluated by the Natural processor *after* the rows have been selected according to the descriptor value(s) specified in the search criterion.

### Processing Limit

If a processing limit is specified as a constant integer number, for example, READ ( 5 ), in the SQL string generated, the value that defines the limitation will be translated into the clause

```
FETCH FIRST integer ROWS ONLY
```

### Cursors for DB2 Clauses

Natural for DB2 uses insensitive scrollable cursors to process the following READ statement:

```
READ .. [ IN ] [ LOGICAL ] VARIABLE/DYNAMIC operand5 [ SEQUENCE ]
```

Natural for DB2 uses insensitive scrollable cursors to process the READ statement below. If relating to a Positioned UPDATE or Positioned DELETE statement, Natural for DB2 uses insensitive static cursors.

```
READ .. [ IN ] [ PHYSICAL ] DESCENDING/VARIABLE/DYNAMIC operand5 [ SEQUENCE ]
```

*operand5*

Value A will be translated into a FETCH FIRST/NEXT DB2 access, and value D into a FETCH LAST/PRIOR DB2 access.

#### Note:

Natural for DB2 supports DB2 multiple row processing on behalf of the MULTIFETCH clause of the READ statement.

### READ when Using the File Server

As far as the file server is concerned there are no programming restrictions with selection statements. It is, however, recommended to make yourself familiar with its functionality considering performance and file server space requirements.

## STORE

The Natural native DML statement STORE is used to add a row to a DB2 table. The STORE statement corresponds to the SQL statement INSERT.



Example:

The Natural native DML statement:

```
STORE RECORD IN EMPLOYEES
  WITH PERSONNEL_ID = '2112'
      NAME           = 'LIFESON'
      FIRST_NAME     = 'ALEX'
```

Equivalent Natural SQL statement:

```
INSERT INTO EMPLOYEES (PERSONNEL_ID, NAME, FIRST_NAME)
  VALUES ('2112', 'LIFESON', 'ALEX')
```

The PASSWORD, CIPHER and USING/GIVING NUMBER clauses of the STORE statement cannot be used.

## UPDATE

The Natural native DML statement UPDATE updates a row in a DB2 table which has been read with a preceding FIND, READ, or SELECT statement. It corresponds to the SQL statement UPDATE WHERE CURRENT OF *cursor-name* (Positioned UPDATE), which means that only the row which was read last can be updated.

### UPDATE when Using the File Server

If a row rolled out to the file server is to be updated, Natural automatically rereads the original row from the database to compare it with its image stored in the file server. If the original row has not been modified in the meantime, the UPDATE operation is performed. With the next terminal I/O, the transaction is terminated and the row is definitely updated on the database.

If the UPDATE operates on a scrollable cursor, the row on the file server and the row in the base table are updated. If the row no longer qualifies for the search criteria of the related SELECT statement after the update, the row is marked as UPDATE hole on the file server.

However, if any modification is detected, the row will not be updated and Natural issues the NAT3703 error message for non-scrollable cursors.

If the UPDATE operates on a scrollable cursor, Natural for DB2 simulates SQLCODE -224 THE RESULT TABLE DOES NOT AGREE WITH THE BASE TABLE USING for DB2 compliance.

If the UPDATE operates on a scrollable cursor and the row has become a hole, Natural for DB2 simulates SQLCODE -222 AN UPDATE OR DELETE OPERATION WAS ATTEMPTED AGAINST A HOLE.

Since an UPDATE statement requires rereading a single row by Natural, a unique index must be available for this table. All columns which comprise the unique index must be part of the corresponding Natural view.

### UPDATE with FIND/READ

As explained with the Natural native DML statement FIND, Natural translates a FIND statement into an SQL SELECT statement. When a Natural program contains a DML UPDATE statement, this statement is translated into an SQL UPDATE statement and a FOR UPDATE OF clause is added to the SELECT statement.

Example:

```
FIND EMPLOYEES WITH SALARY < 5000
  ASSIGN SALARY = 6000
  UPDATE
```

Natural would translate the above Natural statements into SQL and assign a cursor name (for example, `CURSOR1`) as follows:

```
DECLARE CURSOR1 CURSOR FOR
SELECT SALARY FROM EMPLOYEES WHERE SALARY < 5000
  FOR UPDATE OF SALARY
UPDATE EMPLOYEES SET SALARY = 6000
  WHERE CURRENT OF CURSOR1
```

Both the `SELECT` and the `UPDATE` statement refer to the same cursor.

Due to DB2 logic, a column (field) can only be updated if it is contained in the `FOR UPDATE OF` clause; otherwise updating this column (field) is rejected. Natural includes automatically all columns (fields) into the `FOR UPDATE OF` clause which have been modified anywhere in the Natural program or which are input fields as part of a Natural map.

However, an DB2 column is not updated if the column (field) is marked as "not updateable" in the Natural DDM. Such columns (fields) are removed from the `FOR UPDATE OF` list without any warning or error message. The columns (fields) contained in the `FOR UPDATE OF` list can be checked with the `LISTSQL` command.

The Adabas short name in the Natural DDM determines whether a column (field) can be updated.

The following table shows the ranges that apply:

Short-Name Range	Type of Field
AA - N9	non-key field that can be updated
Aa - Nz	non-key field that can be updated
OA - O9	primary key field
PA - P9	ascending key field that can be updated
QA - Q9	descending key field that can be updated
RA - X9	non-key field that cannot be updated
Ra - Xz	non-key field that cannot be updated
YA - Y9	ascending key field that cannot be updated
ZA - Z9	descending key field that cannot be updated
1A - 9Z	non-key field that cannot be updated
1a - 9z	non-key field that cannot be updated

Be aware that a primary key field is never part of a `FOR UPDATE OF` list. A primary key field can only be updated by using a non-cursor `UPDATE` operation (see also Natural SQL `UPDATE` statement in the section *Using Natural SQL Statements*).

A row read with a `FIND` statement that contains a `SORTED BY` clause cannot be updated (due to DB2 limitations as explained with the `FIND` statement). A row read with a `READ LOGICAL` statement cannot be updated either (as explained with the `READ` statement).

If a column is to be updated which is redefined as an array, it is strongly recommended to update the whole column and not individual occurrences; otherwise, results are not predictable. To do so, in reporting mode you can use the `OBTAIN` statement, which must be applied to all field occurrences in the column to be updated. In structured mode, however, all these occurrences must be defined in the corresponding Natural view.

The data locked by an `UPDATE` statement are released when an `END TRANSACTION (COMMIT WORK)` or `BACKOUT TRANSACTION (ROLLBACK WORK)` statement is executed by the program.

**Note:**

If a length indicator field or `NULL` indicator field is updated in a Natural program without updating the field (column) it refers to, the update of the column is not generated for DB2 and thus no updating takes place.

### UPDATE with SELECT

In general, the Natural native DML statement `UPDATE` can be used in both structured and reporting mode. However, after a `SELECT` statement, only the syntax defined for Natural structured mode is allowed:

```
UPDATE [RECORD] [IN] [STATEMENT] [(r)]
```

This is due to the fact that in combination with the `SELECT` statement, the Natural native DML `UPDATE` statement is only allowed in the special case of:

```
...
SELECT ...
  INTO VIEW view-name
  ...
```

Thus, only a whole Natural view can be updated; individual columns (fields) cannot.

Example:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE

SELECT *
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE NAME LIKE 'S%'

  IF NAME = 'SMITH'
    ADD 1 TO AGE
  UPDATE
  END-IF

END-SELECT
...
```

In combination with the Natural native DML UPDATE statement, any other form of the SELECT statement is rejected and an error message is returned.

In all other respects, the Natural native DML UPDATE statement can be used with the SELECT statement in the same way as with the Natural FIND statement.

## Using Natural SQL Statements

This section covers points you have to consider when using Natural SQL statements with DB2. These DB2-specific points partly consist in syntax enhancements which belong to the Extended Set of Natural SQL syntax. The Extended Set is provided in addition to the Common Set to support database-specific features; see *Common Set and Extended Set* in the section *SQL Statements* in the *Natural Statements* documentation.

For information on logging SQL statements contained in a Natural program, refer to *DBLOG Trace Screen for SQL Statements* in the *DBLOG Utility* documentation.

Below is information on the following Natural SQL statements and on common syntactical items:

- Syntactical Items Common to Natural SQL Statements
- CALLDDBPROC - SQL
- COMMIT - SQL
- DELETE - SQL
- INSERT - SQL
- MERGE - SQL
- PROCESS SQL
- READ RESULT SET - SQL
- ROLLBACK - SQL
- SELECT - SQL
- UPDATE - SQL

### Syntactical Items Common to Natural SQL Statements

The following common syntactical items are either DB2-specific and do not conform to the standard SQL syntax definitions (that is, to the Common Set of Natural SQL syntax) or impose restrictions when used with DB2 (see also *SQL Statements* in the *Natural Statements* documentation).

Below is information on the following common syntactical items:

- atom
- comparison
- factor
- scalar-function

- column-function
- scalar-operator
- special-register
- units
- case-expression

### **atom**

An atom can be either a parameter (that is, a Natural program variable or host variable) or a constant. When running dynamically, however, the use of host variables is restricted by DB2. For further details, refer to the relevant DB2 literature by IBM.

### **comparison**

The comparison operators specific to DB2 belong to the Natural Extended Set. For a description, refer to *Comparison Predicate in Search Condition*, *Natural SQL Statements* in the *Natural Statements* documentation.

### **factor**

The following factors are specific to DB2 and belong to the Natural SQL Extended Set:

<i>special-register</i> <i>scalar-function (scalar-expression, ...)</i> <i>scalar-expression unit</i> <i>case-expression</i>
---

### **scalar-function**

A scalar function is a built-in function that can be used in the construction of scalar computational expressions. Scalar functions are specific to DB2 and belong to the Natural SQL Extended Set.

The scalar functions Natural for DB2 supports are listed below in alphabetical order:

A - H	I - R	S - Z
ABS	IDENTITY_VAL_LOCAL	SCORE
ABSVAL	IFNULL	SECOND
ACOS	INSERT	SIGN
ADD_MONTHS	INTEGER	SIN
ASIN	JULIAN_DAY	SINH
ASCII CHR	LAST_DAY	SMALLINT
ASCII_STR	LCASE	SOAPHTTTPC
ATAN	LEFT	SOAPHTTTPV
ATAN2	LENGTH	SOAPHTTTPNC
ATANH	LN	SOAPHTTTPNV
BIGINT	LOCATE	SOUNDEX
BINARY	LOCATE_IN_STRING	SPACE
BLOB	LOG	SQRT
CCSID_ENCODING	LOG10	STRIP
CEIL	LOWER	SUBSTR
CEILING	LPAD	SUBSTRING
CHAR	LTRIM	TAN
CHARACTER_LENGTH	MAX	TANH
CLOB	MICROSECOND	TIME
COALESCE	MIDNIGHT_SECONDS	TIMESTAMP
COLLATION_KEY	MIN	TIMESTAMPADD
COMPARE_DECFLOAT	MINUTE	TIMESTAMP_FORMAT
CONCAT	MOD	TIMESTAMP_ISO
CONTAINS	MONTH	TO_CHAR
COS	MONTHS_BETWEEN	TO_DATE
COSH	MQPUBLISH	TOTALORDER
DATE	MQPUBLISHXML	TRANSLATE
DAY	MQREAD	TRUNC
DAYOFMONTH	MQREADCLOB	TRUNC_TIMESTAMP
DAYOFWEEK	MQREADXML	TRUNCATE
DAYOFWEEK_ISO	MQRECEIVE	UCASE
DAYOFYEAR	MQRECEIVECLOB	UNICODE
DAYS	MQRECEIVEXML	UNICODE_STR
DBCLOB	MQSEND	UNISTR
DEC	MQSENDXML	UPPER
DECFLOAT	MQSENDXMLFILE	VALUE
DECFLOAT_SORTKEY	MQSENDXMLFILECLOB	VARBINARY
DECIMAL	MQSUBSCRIBE	VARCHAR
DECRYPT_BIT	MQUNSUBSCRIBE	VARCHAR_FORMAT
DECRYPT_CHAR	MULTIPLY_ALT	VARGRAPHIC
DECRYPT_DB	NEXT_DAY	WEEK
DEGREES	NORMALIZE_DECFLOAT	WEEK_ISO
DIFFERENCE	NORMALIZE_STRING	XMLATTRIBUTES
DIGITS	NULLIF	XMLCONCAT
DOUBLE	OVERLAY	XMLCOMMENT
DOUBLE_PRECISION	POSSTR	XMLDOCUMENT
DSN_XMLVALIDATE	POWER	XMLELEMENT
EBCDIC CHR	QUANTIZE	XMLFOREST
EBCDIC_STR	QUARTER	XMLNAMESPACES
ENCRYPT_TDES	RADIANS	XMLPARSE
ENCRYPT	RAISE_ERROR	XMLPI
EXP	RAND	XMLQUERY
EXTRACT	REAL	XMLSERIALIZE
FLOAT	REPEAT	XMLTEXT
FLOOR	REPLACE	YEAR
GRAPHIC	RID	
GENERATE_UNIQUE	RIGHT	
GETHINT	ROUND	
GETVARIABLE	ROUND_TIMESTAMP	
HEX	ROWID	
HOURL	RPAD	
	RTRIM	

Each scalar function is followed by one or more scalar expressions in parentheses. The number of scalar expressions depends upon the scalar function. Multiple scalar expressions must be separated from one another by commas.

Example:

```
SELECT NAME
      INTO NAME
      FROM SQL-PERSONNEL
      WHERE SUBSTR ( NAME, 1, 3 ) = 'Fri'
      ...
```

### **column-function**

A column function returns a single-value result for the argument it receives. The argument is a set of like values, such as the values of a column. Column functions are also called aggregating functions.

The following column functions conform to standard SQL. They are not specific to DB2:

```
AVG
COUNT
MAX
MIN
SUM
```

The following column functions do not conform to standard SQL. They are specific to DB2 and belong to the Natural SQL Extended Set.

```
COUNT_BIG
CORRELATION
COVARIANCE
COVARIANCE_SAMP
STDDEV
STDDEV_POP
STDDEV_SAMP
VAR
VAR_POP
VAR_SAMP
VARIANCE
VARIANCE_SAMP
XMLAGG
```

### **scalar-operator**

The concatenation operator (CONCAT or ||) does not conform to standard SQL. It is specific to DB2 and belongs to the Natural Extended Set.

### **special-register**

The following special registers do not conform to standard SQL. They are specific to DB2 and belong to the Natural SQL Extended Set:

CURRENT APPLICATION ENCODING SCHEME  
CURRENT CLIENT\_ACCNTG  
CURRENT CLIENT\_APPLNAME  
CURRENT CLIENT\_USERID  
CURRENT CLIENT\_WRKSTNNAME  
CURRENT DATE  
CURRENT\_DATE  
CURRENT DEBUG MODE  
CURRENT DECFLOAT ROUNDING MODE  
CURRENT DEGREE  
CURRENT FUNCTION PATH  
CURRENT\_LC\_CTYPE  
CURRENT LC\_CTYPE  
CURRENT LOCALE LC\_CTYPE  
CURRENT OPTIMIZATION HINT  
CURRENT PACKAGESET  
CURRENT\_PATH  
CURRENT PRECISION  
CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION  
CURRENT\_MEMBER  
CURRENT PACKAGE PATH  
CURRENT REFRESH AGE  
CURRENT ROUTINE VERSION  
CURRENT SCHEMA  
CURRENT RULES  
CURRENT SQLID  
CURRENT SERVER  
CURRENT TIME  
CURRENT\_TIME  
CURRENT TIMESTAMP  
CURRENT TIMEZONE  
CURRENT\_TIMEZONE USER

A reference to a special register returns a scalar value.

Using the command `SET CURRENT SQLID`, the creator name of a table can be substituted by the current SQLID. This enables you to access identical tables with the same table name but with different creator names.

## **units**

Units, also called "durations", are specific to DB2 and belong to the Natural SQL Extended Set.

The following units are supported:

DAY  
DAYS  
HOUR  
HOURS  
MICROSECOND  
MICROSECONDS  
MINUTE



MINUTES  
 MONTH  
 MONTHS  
 SECOND  
 SECONDS  
 YEAR  
 YEARS

### case-expression

<pre> CASE { searched-when-clause ... } [ ELSE { NULL       simple-when-clause           } ] END       scalar expression </pre>
---

*Case-expressions* do not conform to standard SQL and are therefore supported by the Natural SQL Extended Set only.

### Example:

```

DEFINE DATA LOCAL
  01 #EMP
  02 #EMPNO (A10)
  02 #FIRSTNME (A15)
  02 #MIDINIT (A5)
  02 #LASTNAME (A15)
  02 #EDLEVEL (A13)
  02 #INCOME (P7)
  END-DEFINE
SELECT EMPNO, FIRSTNME, MIDINIT, LASTNAME,
  (CASE WHEN EDLEVEL < 15 THEN 'SECONDARY'
    WHEN EDLEVEL < 19 THEN 'COLLEGE'
    ELSE 'POST GRADUATE'
  END ) AS EDUCATION, SALARY + COMM AS INCOME
INTO
#EMPNO, #FIRSTNME, #MIDINIT, #LASTNAME,
#EDLEVEL, #INCOME
FROM DSN8510-EMP
WHERE (CASE WHEN SALARY = 0 THEN NULL
  ELSE SALARY / COMM
  END ) > 0.25

DISPLAY #EMP
END-SELECT
END

```

## CALLDBPROC - SQL

The Natural SQL statement CALLDBPROC is used to call DB2 stored procedures. It supports the result set mechanism of DB2 and it enables you to call DB2 stored procedures. For further details and statement syntax, see CALLDBPROC in the Natural *Statements* documentation.

The following topics are covered below:

- Static and Dynamic Execution
- Result Sets
- List of Parameter Data Types
- CALLMODE=NATURAL

- Example of CALLDBPROC/READ RESULT SET

## Static and Dynamic Execution

If the CALLDBPROC statement is executed dynamically, all parameters and constants are mapped to the variables of the following DB2 SQL statement:

```
CALL :hv USING DESCRIPTOR :sqlda statement
```

:hv denotes a host variable containing the name of the procedure to be called and :sqlda is a dynamically generated sqlda describing the parameters to be passed to the stored procedure.

If the CALLDBPROC statement is executed statically, the constants of the CALLDBPROC statement are also generated as constants in the generated assembler SQL source for the DB2 precompiler.

## Result Sets

If the SQLCODE created by the CALL statement indicates that there are result sets (SQLCODE +466 and +464), Natural for DB2 runtime executes a DESCRIBE PROCEDURE :hv INTO :sqlda statement in order to retrieve the result set locator values of the result sets created by the invoked stored procedure. These values are put into the RESULT SETS variables specified in the CALLDBPROC statement. Each RESULT SETS variable specified in a CALLDBPROC for which no result set locator value is present is reset to zero. The result set locator values can be used to read the result sets by means of the READ RESULT SET statement as long as the database transaction which created the result set has not yet issued a COMMIT or ROLLBACK.

If the result set was created by a cursor WITH HOLD, the result set locator value remains valid after a COMMIT operation.

Unlike other Natural SQL statements, CALLDBPROC enables you (optionally!) to specify an SQLCODE variable following the GIVING keyword which will contain the SQLCODE of the underlying CALL statement. If GIVING is specified, it is up to the Natural program to react on the SQLCODE (error message NAT3700 is not issued by the runtime).

## List of Parameter Data Types

Below are the parameter data types supported by the CALLDBPROC statement:

Natural Format/Length	DB2 Data Type
An	CHAR ( n )
B2	SMALLINT
B4	INT
Bn (n = not equal 2 or 4)	CHAR ( n )
F4	REAL
F8	DOUBLE PRECISION
I2	SMALLINT
I4	INT
Nnn.m	NUMERIC ( nn+m , m )
Pnn.m	NUMERIC ( nn+m , n )
Gn	GRAPHIC ( n )
An/1:m	VARCHAR ( n*m )
D	DATE
T	TIME  <b>Note:</b> The Natural format T has a wider data range than the equivalent DB2 TIME data type. Compared with DB2 TIME, in addition, the Natural T variable has a date fraction (year, month, day) and the tenths of a second. As a result, when converting a Natural T variable into a DB2 TIME value, Natural for DB2 cuts off the date fraction and the tenths of a second part. When converting DB2 TIME into Natural T format, the date fraction is reset to 0000-01-02 and the tenths of a second part is reset to 0 in Natural.

## CALLMODE=NATURAL

This parameter is used to invoke Natural stored procedures defined with `PARAMETER STYLE GENERAL/WITH NULL`.

If the `CALLMODE=NATURAL` parameter is specified, an additional parameter describing the parameters passed to the Natural stored procedure is passed from the client, that is, caller, to the server, that is, the Natural for DB2 server stub. The parameter is the Stored Procedure Control Block (STCB; see also *STCB Layout* in *PARAMETER STYLE* in the section *Processing Natural Stored Procedures and UDFs*) and has the format `VARCHAR` from the viewpoint of DB2. Therefore, every Natural stored procedure defined with `PARAMETER STYLE GENERAL/WITH NULL` has to be defined with the `CREATE PROCEDURE` statement by using this `VARCHAR` parameter as the first in its `PARMLIST` row.

From the viewpoint of the caller, that is, the Natural program, and from the viewpoint of the stored procedure, that is, Natural subprogram, the STCB is invisible. It is passed as first parameter by the Natural for DB2 runtime and it is used as on the server side to build the copy of the passed data in the Natural

thread and the corresponding CALLNAT statement. Additionally, this parameter serves as a container for error information created during execution of the Natural stored procedure by the Natural runtime. It also contains information on the library where you are logged on and the Natural subprogram to be invoked.

### Example of CALLDBPROC/READ RESULT SET

Below is a sample program for issuing CALLDBPROC and READ RESULT SET statements:

```

DEFINE DATA LOCAL
  1 ALPHA          (A8)
  1 NUMERIC        (N7.3)
  1 PACKED         (P9.4)
  1 VCHAR          (A20/1:5) INIT    <'DB25SGCP'>
  1 INTEGER2       (I2)
  1 INTEGER4       (I4)
  1 BINARY2        (B2)
  1 BINARY4        (B4)
  1 BINARY12       (B12)
  1 FLOAT4         (F4)
  1 FLOAT8         (F8)
  1 INDEX-ARRAY   (I2/1:11)
  1 INDEX-ARRAY1  (I2)
  1 INDEX-ARRAY2  (I2)
  1 INDEX-ARRAY3  (I2)
  1 INDEX-ARRAY4  (I2)
  1 INDEX-ARRAY5  (I2)
  1 INDEX-ARRAY6  (I2)
  1 INDEX-ARRAY7  (I2)
  1 INDEX-ARRAY8  (I2)
  1 INDEX-ARRAY9  (I2)
  1 INDEX-ARRAY10 (I2)
  1 INDEX-ARRAY11 (I2)
  1 #RESP         (I4)
  1 #RS1          (I4) INIT <99>
  1 #RS2          (I4) INIT <99>
LOCAL
  1 V1 VIEW OF SYSIBM-SYSTABLES
  2 NAME
  1 V2 VIEW OF SYSIBM-SYSPROCEDURES
  2 PROCEDURE
  2 RESULT_SETS
  1 V (I2) INIT <99>
END-DEFINE
CALLDBPROC 'DAEFDB25.SYSPROC.SNGSTPC' DSN8510-EMP
  ALPHA INDICATOR :INDEX-ARRAY1
  NUMERIC INDICATOR :INDEX-ARRAY2
  PACKED INDICATOR :INDEX-ARRAY3
  VCHAR(*) INDICATOR :INDEX-ARRAY4
  INTEGER2 INDICATOR :INDEX-ARRAY5
  INTEGER4 INDICATOR :INDEX-ARRAY6
  BINARY2 INDICATOR :INDEX-ARRAY7
  BINARY4 INDICATOR :INDEX-ARRAY8
  BINARY12 INDICATOR :INDEX-ARRAY9
  FLOAT4 INDICATOR :INDEX-ARRAY10
  FLOAT8 INDICATOR :INDEX-ARRAY11
  RESULT SETS #RS1 #RS2
  CALLMODE=NATURAL
  READ (10) RESULT SET #RS2 INTO VIEW V2 FROM SYSIBM-SYSTABLES
  WRITE 'PROC F RS  :' PROCEDURE 50T RESULT_SETS
END-RESULT
END

```

## COMMIT - SQL

The Natural SQL COMMIT statement indicates the end of a logical transaction and releases all DB2 data locked during the transaction. All data modifications are made permanent.

COMMIT is a synonym for the Natural native DML statement END TRANSACTION as described in the section *Using Natural Native DML Statements*.

No transaction data can be provided with the COMMIT statement.

If this command is executed from a Natural stored procedure or user-defined function (UDF), Natural for DB2 does not execute the underlying commit operation. This allows the Natural stored procedure or UDF to commit updates against non DB2 databases.

Under CICS, the COMMIT statement is translated into an EXEC CICS SYNCPOINT command. If the file server is used, an implicit end-of-transaction is issued after each terminal I/O. This is due to CICS-specific transaction processing in pseudo-conversational mode, see *Natural for DB2 under CICS*.

Under IMS TM, the COMMIT statement is not translated into an IMS CHECKPOINT command, but is ignored. An implicit end-of-transaction is issued after each terminal I/O. This is due to IMS TM-specific transaction processing, see *Natural for DB2 under IMS TM*.

Unless when used in combination with the WITH HOLD clause (see *SELECT - Cursor-Oriented* in the *Natural Statements* documentation), a COMMIT statement must not be placed within a database loop, since all cursors are closed when a logical unit of work ends. Instead, it has to be placed outside such a loop or after the outermost loop of nested loops.

If an external program written in another standard programming language is called from a Natural program, this external program must not contain its own COMMIT command if the Natural program issues database calls, too. The calling Natural program must issue the COMMIT statement on behalf of the external program.

For further details and statement syntax, see COMMIT - SQL in the *Natural Statements* documentation.

## DELETE - SQL

Both the cursor-oriented or Positioned DELETE, and the non-cursor or Searched DELETE statements are supported as part of Natural SQL; the functionality of the Positioned DELETE statement corresponds to that of the Natural DML DELETE statement. For further details and statement syntax, see DELETE in the *Natural Statements* documentation.

With DB2, a table name in the FROM clause of a Searched DELETE statement can be assigned a *correlation-name*. This does not correspond to the standard SQL syntax definition and therefore belongs to the Natural SQL Extended Set.

The Searched DELETE statement must be used, for example, to delete a row from a self-referencing table, since with self-referencing tables a Positioned DELETE is not allowed by DB2.

For further details and statement syntax, see DELETE - SQL in the *Natural Statements* documentation.

## INSERT - SQL

The Natural SQL `INSERT` statement is used to add one or more new rows to a table.

Since the `INSERT` statement can contain a select expression, all the DB2-specific common syntactical items described above apply.

For further details and statement syntax, see *INSERT - SQL* in the *Natural Statements* documentation.

## MERGE - SQL

The `MERGE` statement is a hybrid SQL statement consisting of an `UPDATE` component and an `INSERT` component. It allows you either to insert a row into a DB2 table or to update a row of a DB2 table if the input data matches an already existing row of a table.

The `MERGE` statement belongs to the SQL Extended Set.

For further details and statement syntax, see *MERGE - SQL* in the *Natural Statements* documentation.

## PROCESS SQL

The Natural `PROCESS SQL` statement is used to issue SQL statements to the underlying database. The statements are specified in a *statement-string*, which can also include constants and parameters. The set of statements which can be issued is also referred to as Flexible SQL and comprises those statements which can be issued with the SQL statement `EXECUTE`.

In addition, Flexible SQL includes the following DB2-specific statements:

```
CALL
CONNECT
GET DIAGNOSTICS
SET APPLICATION ENCODING SCHEME
SET CONNECTION
SET CURRENT DEGREE
SET CURRENT LC_CTYPE
SET CURRENT OPTIMIZATION HINT
SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
SET CURRENT PACKAGE PATH
SET CURRENT PACKAGESET
SET CURRENT PATH
SET CURRENT PRECISION
SET CURRENT REFRESH AGE
SET CURRENT RULES
SET CURRENT SCHEMA
SET CURRENT SQLID
SET ENCRYPTION PASSWORD
SET host-variable=special-register
RELEASE
```

### Notes:

1. SQL statements issued by `PROCESS SQL` are skipped during static generation. Thus they are always executed dynamically via `NDBIOMO`.
2. To avoid transaction synchronization problems between the Natural environment and DB2, the `COMMIT` and `ROLLBACK` statements must not be used within `PROCESS SQL`.

For further details and statement syntax, see `PROCESS SQL` in the *Natural Statements* documentation.

## READ RESULT SET - SQL

The Natural SQL `READ RESULT SET` statement reads a result set created by a Natural stored procedure that was invoked by a `CALLDBPROC` statement. For details on how to specify the scroll direction by using the variable *scroll-hv*, see the `SELECT` statement.

For further details and statement syntax, see `READ RESULT SET` in the *Natural Statements* documentation.

## ROLLBACK - SQL

The Natural SQL `ROLLBACK` statement undoes all database modifications made since the beginning of the last logical transaction. Logical transactions can start either after the beginning of a session or after the last `COMMIT/END TRANSACTION` or `ROLLBACK/BACKOUT TRANSACTION` statement. All records held during the transaction are released.

For further details and statement syntax, see `ROLLBACK -SQL` in the *Natural Statements* documentation.

`ROLLBACK` is a synonym for the Natural statement `BACKOUT TRANSACTION` as described in the section *Using Natural Native DML Statements*.

If this command is executed from a Natural stored procedure or user-defined function (UDF), Natural for DB2 executes the underlying rollback operation. This sets the caller into a must-rollback state. If this command is executed by Natural error processing (implicit `ROLLBACK`), Natural for DB2 does not execute the underlying rollback operation, thus allowing the caller to receive the original Natural error.

Under CICS, the `ROLLBACK` statement is translated into an `EXEC CICS ROLLBACK` command. However, if the file server is used, only changes made to the database since the last terminal I/O are undone. This is due to CICS-specific transaction processing in pseudo-conversational mode, see *Natural for DB2 under CICS*.

Under IMS TM, the `ROLLBACK` statement is translated into an IMS Rollback (`ROLB`) command. However, only changes made to the database since the last terminal I/O are undone. This is due to IMS TM-specific transaction processing, see *Natural for DB2 under IMS TM*.

As all cursors are closed when a logical unit of work ends, a `ROLLBACK` statement must not be placed within a database loop; instead, it has to be placed outside such a loop or after the outermost loop of nested loops.

If an external program written in another standard programming language is called from a Natural program, this external program must not contain its own `ROLLBACK` command if the Natural program issues database calls, too. The calling Natural program must issue the `ROLLBACK` statement on behalf of the external program.

## SELECT - SQL

The Natural SQL `SELECT` statement supports both the cursor-oriented selection, which is used to retrieve an arbitrary number of rows and the non-cursor selection (Singleton `SELECT`), which retrieves at most one single row.

For full details and statement syntax, see `SELECT -SQL` in the *Natural Statements* documentation.

### SELECT - Cursor-Oriented

Like the Natural native DML `FIND` statement, the cursor-oriented `SELECT` statement is used to select a set of rows (records) from one or more DB2 tables, based on a search criterion. Since a database loop is initiated, the loop must be closed by a `LOOP` statement (in reporting mode) or by an `END-SELECT` statement (in structured mode). With this construction, Natural uses the same loop processing as with the `FIND` statement. In addition, no cursor management is required from the application program; it is automatically handled by Natural.

For further details and syntax, see *SELECT - SQL, Syntax 1 - Cursor-Oriented Selection* in the *Natural Statements* documentation.

### SELECT SINGLE - Non-Cursor-Oriented

The Natural SQL statement `SELECT SINGLE` provides the functionality of a non-cursor selection (Singleton `SELECT`); that is, a select expression that retrieves at most one row without using a cursor.

Since DB2 supports the Singleton `SELECT` command in static SQL only, in dynamic mode, the Natural `SELECT SINGLE` statement is executed in the same way as a set-level `SELECT` statement, which results in a cursor operation. However, Natural checks the number of rows returned by DB2. If more than one row is selected, a corresponding error message is returned.

For further details and syntax, see *SELECT - SQL, Syntax 2 - Non-Cursor Selection* in the *Natural Statements* documentation.

## UPDATE - SQL

Both the cursor-oriented or Positioned `UPDATE` and the non-cursor or Searched `UPDATE` statements are supported as part of Natural SQL. Both of them reference either a table or a Natural view.

With DB2, the name of a table or Natural view to be referenced by a Searched `UPDATE` can be assigned a *correlation-name*. This does not correspond to the standard SQL syntax definition and therefore belongs to the Natural Extended Set.

The Searched `UPDATE` statement must be used, for example, to update a primary key field, since DB2 does not allow updating of columns of a primary key by using a Positioned `UPDATE` statement.

### Note:

If you use the `SET *` notation, all fields of the referenced Natural view are added to the `FOR UPDATE OF` and `SET` lists. Therefore, ensure that your view contains only fields which can be updated; otherwise, a negative `SQLCODE` is returned by DB2.



For further details and syntax, see *UPDATE - SQL* in the *Natural Statements* documentation.

## Using Natural System Variables

When used with DB2, there are restrictions and/or special considerations concerning the following Natural system variables:

- \*ISN
- \*NUMBER
- \*ROWCOUNT

For information on restrictions and/or special considerations, refer to the section *Database-Specific Information* in the corresponding system variable documentation.

## Multiple Row Processing

This section describes the multiple row functionality for DB2 databases.

You have to operate against DB2 for z/OS Version 8 or higher to use these features.

Natural for DB2 provides two kinds of multiple row processing features:

- **Standard multiple row processing**
- This feature does not influence the program logic. Although the Natural native DML and Natural SQL DML provide clauses for specification of the multi-fetch-factor, the Natural program operates with one database row and from the program point of view only one row is received from or is send to the database.
- **Advanced multiple row processing**

This feature is only available with Natural SQL DML and has a lot of impact on the program logic, as it allows the retrieval of multiple rows from the database into the program storage by a single Natural SQL `SELECT` statement into a set of arrays. Additionally it is possible to insert multiple rows into the database from a set of arrays by the Natural SQL `INSERT` statement.

Below is information on the following topics:

- Purpose of Multi-Fetch Feature (Standard)
- Considerations for Multi-Fetch Usage (Standard)
- Size of the Multi-Fetch Buffer (Standard)
- Support of TEST DBLOG Q (Standard)
- Multiple Rows to Program (Advanced)

- Multiple Rows from Program (Advanced)

## Purpose of Multi-Fetch Feature (Standard)

In standard mode, Natural does not read multiple records with a single database call; it always operates in a one-record-per-fetch mode. This kind of operation is solid and stable, but can take some time if a large number of database records are being processed.

To improve the performance of those programs, you can use the Multi-Fetch Clause in the Natural DML FIND, READ or HISTOGRAM statements. This allows you to specify the number of records read per database access.

$\left[ \begin{array}{l} \text{FIND} \\ \text{READ} \\ \text{HISTOGRAM} \end{array} \right] \left[ \text{MULTI-FETCH} \left\{ \begin{array}{l} \text{ON} \\ \text{OFF} \\ \text{OF } \textit{multi-fetch-factor} \end{array} \right\} \right]$
--

Where the *multi-fetch-factor* is either a constant or a variable with a format integer (I4).

To improve the performance of the Natural SQL SELECT statements, you can use the WITH ROWSET POSITIONING FOR Clause to specify a multi-fetch-factor.

$\left[ \text{WITH ROWSET POSITIONING FOR} \left\{ \begin{array}{l} [:] \textit{row\_hv} \\ \textit{integer} \end{array} \right\} \text{ROWS} \right]$
--

At statement execution time, the runtime checks if a *multi-fetch-factor* greater than 1 is supplied for the database statement.

If the *multi-fetch-factor* is

less than or equal to 1	the database call is continued in the usual one-record-per-access mode.
greater than 1	the database call is prepared dynamically to read multiple records (e.g. 10) with a single database access into an auxiliary buffer (multi-fetch buffer). If successful, the first record is transferred into the underlying data view. Upon the execution of the next loop, the data view is filled directly from the multi-fetch buffer, without database access. After all records are fetched from the multi-fetch buffer, the next loop results in the next record set being read from the database. If the database loop is terminated (either by end-of-records, ESCAPE, STOP, etc.), the content of the multi-fetch buffer is released.

## Considerations for Multi-Fetch Usage (Standard)

- The program does not receive "fresh" records from the database for every loop, but operates with images retrieved at the most recent multi-fetch access.
- If a dynamic direction change (IN DYNAMIC . . . SEQUENCE) is coded for a Natural DML READ or HISTOGRAM statement, the multi-fetch feature is not possible and leads to a corresponding syntax error at compilation.

- The size occupied by a database loop in the multi-fetch buffer is determined according to the rule:

$$\begin{aligned} & \text{header} + \text{sqldaheader} + \text{columns} * (\text{sqlvar} + \text{lise}) + \text{mf} * (\text{udind} + \text{sum}(\text{collen}) + \text{sum}(\text{LF}(\text{columns}) + \text{sum}(\text{nullind})) \\ & \equiv \\ & 32 + 16 + \text{columns} * (44 + 12) + \text{mf} * (1 + \text{sum}(\text{collen}) + \text{sum}(\text{LF}(\text{column})) + \text{sum}(2)) \end{aligned}$$

where

- header denotes the length of the header of an entry in the DB2 multifetch buffer, that is, 32
- sqldaheader denotes the length of the header of a sqlda, that is, 16
- columns denotes the number of receiving fields of a SQL request
- sqlvar denotes the length of a sqlvar, that is, 44
- lise denotes the length of a Natural for DB2 specific sqlvar extension
- mf denotes the multifetch factor, that is, the number of rows fetched by one database call
- collen denotes the length of the receiving field
- LF(column) denotes the size of the length field of the receiving field, that is, 0 for fixed length fields, 2 for variable length fields, and 4 for large object columns (LOBs)
- nullind denotes the length of a null indicator, that is, 2

## Size of the Multi-Fetch Buffer (Standard)

The multifetch buffer is released at terminal i/o in pseudo conversational mode. Therefore there is no size limitation for the DB2 multifetch buffer (DB2SIZE6). The buffer will be automatically enlarged if necessary.

## Support of TEST DBLOG Q (Standard)

When multi-fetch is used, real database calls are only submitted to get a new set of records.

The TEST DBLOG Q facility is also called from the Natural for DB2 multi fetch handler for every rowset fetch from DB2 and for every record moved from the multi fetch buffer to the program storage. The events are distinguished by the literal MULTI FETCH . . . and <BUFF FETCH . . .

### Example: TEST DBLOG List Break-Out

```

10:51:57          ***** NATURAL Test Utilities *****          2006-01-27
User HGK          - DBLOG Trace -          Library NDB42
M No  R  SQL Statement (truncated)  CU SN SREF M Typ  SQLC/W Program  Line LV
_  1  SELECT EMPNO,FIRSTNME, LASTNAM 01 01 0260 D DB2  MF000001 0260 01
_  2  MULTI FETCH NEX                01 01 0260 D DB2  MF000001 0260 01
_  3  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_  4  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_  5  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_  6  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_  7  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_  8  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_  9  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_ 10  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_ 11  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_ 12  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_ 13  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_ 14  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_ 15  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_ 16  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
_ 17  <BUFF FETCH NEX                00 00 0260 D DB2  MF000001 0260 01
Command ==>

```

where column **No** represents the following:

1	is a open cursor DB2 call.
2	is a "real" database call that reads a set of records via multi-fetch (see <b>MULTI FETCH NEX</b> in column <b>SQL Statement</b> ).
3-17	are "no real" database calls, but only entries that document that the program has received these records from the multi-fetch buffer (see <b>&lt;BUFF FETCH NEX</b> in column <b>SQL Statement</b> ).

## Multiple Rows to Program (Advanced)

The feature allows programs to retrieve multiple rows from DB2 into arrays.

This feature is only available with the **SELECT** statement.

- Prerequisites
- **DB2ARRAY=ON**
- **INTO** Clause
- **WITH ROWSET POSITIONING** Clause
- **ROWS\_RETURNED** Clause
- Restrictions and Constraints
- File Server Usage and Positioned **UPDATE** and **DELETE**

### Prerequisites

#### To use this feature

1. Set the compiler option **DB2ARRAY=ON** (by using an **OPTIONS** statement or the **COMPOPT** command or the **CMPO** profile parameter).

2. Specify a list of receiving arrays in the INTO Clause of the SELECT statement.
3. Specify the number of rows to be retrieved from the database by a single FETCH operation via the WITH ROWSET POSITIONING Clause.
4. Specify a variable receiving the number of rows retrieved from the database via the ROWS\_RETURNED Clause.

## **DB2ARRAY=ON**

DB2ARRAY=ON is necessary to allow the specification of arrays in the INTO Clause. DB2ARRAY=ON also prevents the usage of arrays as sending or receiving fields for DB2 CHAR/VARCHAR /GRAPHIC/VARGRAPHIC columns. Instead Natural scalar fields with the appropriate length have to be used.

## **INTO Clause**

Each array specified in the INTO Clause has to be contiguous (one occurrence following immediately by another, this is expected by DB2) and has to be one-dimensional. The arrays are filled from the first occurrence (low) to last occurrence (high). The first array occurrences compose the first row of the received rowset, the second array occurrences compose the second row of the received rowset. The array occurrences of the nth index compose the nth row returned from DB2. If an LINDICATOR or INDICATOR Clause is used in the INTO Clause for arrays, the specified length indicators or null indicators have also to be arrays. The number of occurrences of LINDICATOR and INDICATOR arrays have to equal or greater than the number of occurrences of the master array.

## **WITH ROWSET POSITIONING Clause**

The WITH\_ROWSET\_POSITIONING Clause is used to specify the number of rows to be retrieved from the database by one processing cycle. The specified number has to be equal or smaller than the minimum of occurrences of all specified arrays. If a variable, not a constant, is specified the actual content of the variable will be used during each processing cycle. The specified number has to be greater 0 and smaller than 32768.

## **ROWS\_RETURNED Clause**

The ROWS\_RETURNED Clause is used to specify a variable, which will contain the number of rows read from the database during the actual fetch operation. The format of the variable has to be I4.

## **Restrictions and Constraints**

Natural Views: It is not possible to use Natural arrays of views in the INTO clause, that is, the use of keyword VIEW is not possible.

## **File Server Usage and Positioned UPDATE and DELETE**

The purpose of this feature is to reduce the number of database and database interface calls for bulk batch processing. Therefore it is not recommended to use this kind of programming in online CICS or IMS environments, when terminal I/Os occur within open cursor loops; that is, the file server is used. A fortiori it is not possible to perform a Positioned UPDATE or Positioned DELETE statement after terminal I/O.

**Example:**

```


DEFINE DATA LOCAL
01 NAME          (A20/1:10)
01 ADDRESS       (A100/1:10)
01 DATEOFBIRTH  (A10/1:10)
01 SALARY        (P4.2/1:10)
01 L$ADDRESS     (I2/1:10)
01 ROWS          (I4)
01 NUMBER        (I4)
01 INDEX         (I4)
END-DEFINE
OPTIONS DB2ARRY=ON
ASSIGN NUMBER := 10
SEL.
SELECT NAME, ADDRESS , DATEOFBIRTH, SALARY
      INTO :NAME(*), /* <-- ARRAY
           :ADDRESS(*) LINDICATOR :L$ADDRESS(*), /* <-- ARRAY
           :DATEOFBIRTH(1:10), /* <-- ARRAY
           :SALARY(01:10) /* <-- ARRAY
      FROM NAT-DEMO
      WHERE NAME > ' '
      WITH ROWSET POSITIONING FOR :NUMBER ROWS /* <-- ROWS REQ
      ROWS_RETURNED :ROWS /* <-- ROWS RET
IF ROWS > 0
  FOR INDEX = 1 TO ROWS STEP 1
    DISPLAY
      INDEX (EM=99) *COUNTER (SEL.) (EM=99) ROWS (EM=99)
      NAME(INDEX)
      ADDRESS(INDEX) (AL=20)
      DATEOFBIRTH(INDEX)
      SALARY(INDEX)
    END-FOR
  END-IF
END-SELECT
END

```

**Multiple Rows from Program (Advanced)**

The feature allows programs to insert multiple rows into a DB2 table from arrays.

This feature is only available with the Natural SQL INSERT statement.

**Prerequisites**** To use this feature**

1. Set the compiler option DB2ARRY=ON (by using an OPTIONS statement or the COMPOPT command or the CMPO profile parameter).
2. Specify a list of sending arrays in the VALUES Clause of the Natural SQL INSERT statement.
3. Specify the number of rows to be inserted into the database by a single Natural SQL INSERT statement via the FOR *n* ROWS Clause.

**DB2ARRAY=ON**

DB2ARRAY=ON is necessary to allow the specification of arrays in the VALUES Clause. DB2ARRAY=ON also prevents the usage of arrays as sending or receiving fields for DB2 CHAR/VARCHAR /GRAPHIC/VARGRAPHIC columns. Instead Natural scalar fields with the appropriate length have to be used.

**VALUES Clause**

Each array specified in the VALUES Clause has to be contiguous (one occurrence following immediately by another, this is expected by DB2) and has to be one-dimensional. The arrays are read from the first occurrence (low) to last occurrence (high). The first array occurrences compose the first row inserted into the database, the second array occurrences compose the second row inserted into the database. The array occurrences of the nth index compose the nth row inserted into the database. If a LINDICATOR or INDICATOR Clauses are used in the VALUES Clause for arrays, the specified length indicators or null indicators have also to be arrays. The number of LINDICATOR and INDICATOR array occurrences has to be equal or greater than the number of occurrences of the master array.

**FOR n ROWS Clause**

The FOR *n* ROWS Clause is used to specify how many rows are to be inserted into the database table by one INSERT statement. The specified number has to be equal or smaller than the minimum of occurrences of all specified arrays in the VALUES Clause. The specified number has to be greater than 0 and smaller than 32768.

**Restrictions and Constraints**

- **Natural Views**

It is not possible to use Natural arrays of views in the VALUES clause, that is, the use of keyword VIEW is not possible.

- **Static Execution**

Due to DB2 restrictions it is not possible to execute multiple row inserts in static mode. Therefore multiple row inserts are not generated static and are always dynamically prepared and executed by Natural for DB2.

It is not possible to use Natural arrays of views in the INTO clause, that is, the use of keyword VIEW is not possible.

Example:

```
DEFINE DATA LOCAL
01 NAME          (A20/1:10)  INIT  <'ZILLER1','ZILLER2','ZILLER3','ZILLER4'
                                , 'ZILLER5','ZILLER6','ZILLER7','ZILLER8'
                                , 'ZILLER9','ZILLERA'>
01 ADDRESS       (A100/1:10) INIT  <'ANGEL STREET 1','ANGEL STREET 2'
                                , 'ANGEL STREET 3','ANGEL STREET 4'
                                , 'ANGEL STREET 5','ANGEL STREET 6'
                                , 'ANGEL STREET 7','ANGEL STREET 8'
                                , 'ANGEL STREET 9','ANGEL STREET 10'>
01 DATENATD (D/1:10)  INIT  <D'1954-03-27',D'1954-03-27',D'1954-03-27'
                                ,D'1954-03-27',D'1954-03-27',D'1954-03-27'
                                ,D'1954-03-27',D'1954-03-27',D'1954-03-27'>
```

```

                                ,D'1954-03-27'>
01 SALARY      (P4.2/1:10) INIT <1000,2000,3000,4000,5000
                                ,6000,7000,8000,9000,9999>
01 SALARY_N    (N4.2/1:10) INIT <1000,2000,3000,4000,5000
                                ,6000,7000,8000,9000,9999>
01 L$ADDRESS   (I2/1:10) INIT <14,14,14,14,14,14,14,14,14,15>
01 N$ADDRESS   (I2/1:10) INIT <00,00,00,00,00,00,00,00,00,00>
01 ROWS        (I4)
01 INDEX       (I4)
01 V1 VIEW OF NAT-DEMO_ID
02 NAME
02 ADDRESS     (EM=X(20))
02 DATEOFBIRTH
02 SALARY
01 ROWCOUNT  (I4)
END-DEFINE
OPTIONS DB2ARRY=ON                /* <-- ENABLE DB2 ARRAY
ROWCOUNT := 10
INSERT INTO NAT-DEMO_ID
  (NAME,ADDRESS,DATEOFBIRTH,SALARY)
  VALUES
  (:NAME(*),                      /* <-- ARRAY
   :ADDRESS(*)                    /* <-- ARRAY
   INDICATOR :N$ADDRESS(*)        /* <-- ARRAY
   LINDICATOR :L$ADDRESS(*),      /* <-- ARRAY DB2 VCHAR
   :DATENATD(1:10),              /* <-- ARRAY NATURAL DATES
   :SALARY_N(01:10)              /* <-- ARRAY NATURAL NUMERIC
  )
  FOR :ROWCOUNT ROWS
SELECT * INTO VIEW V1 FROM NAT-DEMO_ID WHERE NAME > 'Z'
DISPLAY V1                        /* <-- VERIFY INSERT
END-SELECT
BACKOUT
END

```

## Error Handling

In contrast to the normal Natural error handling, where either an `ON ERROR` statement is used to intercept execution time errors or standard error message processing is performed and program execution is terminated, the enhanced error handling of Natural for DB2 provides an application controlled reaction to the encountered SQL error.

Two Natural subprograms, `NDBERR` and `NDBNOERR`, are provided to disable the usual Natural error handling and to check the encountered SQL error for the returned SQL code. This functionality replaces the `E` function of the `DB2SERV` interface, which is still provided but no longer documented.

For further information on Natural subprograms provided for DB2, see the section *Interface Subprograms*.

Example:

```

DEFINE DATA LOCAL
01 #SQLCODE      (I4)
01 #SQLSTATE     (A5)
01 #SQLCA        (A136)
01 #DBMS        (B1)
END-DEFINE
*
*           Ignore error from next statement
*

```



```
CALLNAT 'NDBNOERR'
*
*       This SQL statement produces an SQL error
*
INSERT INTO SYSIBH-SYSTABLES (CREATOR, NAME, COLCOUNT)
VALUES ('SAG', 'MYTABLE', '3')
*
*       Investigate error
*
CALLNAT 'NDBERR' #SQLCODE #SQLSTATE #SQLCA #DBMS
*
IF #DBMS NE 2                               /* not DB2
  MOVE 3700 TO *ERROR-NR
END-IF
*
DECIDE ON FIRST VALUE OF #SQLCODE
  VALUE 0, 100                               /* successful execution
    IGNORE
  VALUE -803                                 /* duplicate row
    /* UPDATE existing record
    /*
    IGNORE
  NONE VALUE
    MOVE 3700 TO *ERROR-NR
END-DECIDE
*
END
```