SELECT - SQL

This chapter covers the following topics:

- Function
- Syntax Description
- Join Queries
- SELECT - Cursor-Oriented

For an explanation of the symbols used in the syntax diagram, see *Syntax Symbols*.

Belongs to Function Group: *Database Access and Update*

See also the following sections in the *Database Management System Interfaces* documentation:

- *SELECT SINGLE - Non-Cursor-Oriented* in the *Natural for DB2* part.
- *SELECT* in the *Natural for SQL/DS* part.

Function

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

With the `SELECT ... END-SELECT` construction, Natural uses the same database loop processing as with the `FIND` statement.

Syntax Description

Two different structures are possible:

Syntax 1 - Cursor-Oriented Selection

Common Set Syntax

```
SELECT selection INTO    { parameter,...                      } table-expression
                         { VIEW {view-name                    }
                         { [correlation-name]},...            }

[ { UNION      } [ALL] [(]SELECT selection table-expression[)]  ]
[ { EXCEPT     }                                                ]
[ { INTERSECT  }                                                ]

ORDER BY                 { integer          } [        ASC  ]
                         { column-reference }  [        DESC ]
                         { expression       }

statement ...
{  END-SELECT (structured mode only)  }
   LOOP (reporting mode only)
```

(ASC is underlined as the default.)

Extended Set Syntax:

```
[WITH_CTE common-table-expression,...]
SELECT selection INTO        { parameter,...
                             { VIEW {view-name [correlation-name]},...
[ { UNION                    } [ALL] [(]SELECT selection table-expression[)]                  ]
[ { EXCEPT                   }                                                                ]
[ { INTERSECT                }                                                                ]
ORDER BY                     { integer                    } [ ASC  ]
                             { column-reference           } [ DESC ]
                             { expression                 }
                             { INPUT SEQUENCE             }
[OPTIMIZE FOR integer ROWS]
[ WITH                       { CS                         } ]
[                            { RR                         } ]
[                            { UR                         } ]
[                            { RS                         } ]
[                              RS KEEP UPDATE LOCKS         ]
[                              RR KEEP UPDATE LOCKS         ]
QUERYNO integer
[ FETCH FIRST                [ { 1                      } ] { ROW  }             ONLY ]
[                            [ { integer                } ] { ROWS }                  ]
[WITH HOLD]
[WITH RETURN]
[ WITH                       { ASENSITIVE SCROLL          } [:]scroll_hv [GIVING [:] sqlcode ]
[                            { INSENSITIVE SCROLL         }                                  ]
[                            { SENSITIVE STATIC SCROLL    }                                  ]
[                            { SENSITIVE DYNAMIC SCROLL   }                                  ]
[ WITH ROWSET POSITIONING    { [:] row_hv                 } ROWS ]    ROWS_RETURNED ]
[ FOR                        { integer                    }      ]    [:] ret_row   ]
[IF-NO-RECORDS-FOUND-clause]
statement ...
{ END-SELECT (structured mode only) }
{ LOOP       (reporting mode only)  }
```

Syntax Element Description - Syntax 1:

SELECT *selection*	Like the `FIND` statement, a cursor-oriented selection is used to select a set of rows (records) from one or more database tables, based on a search criterion. In addition, no cursor management is required from the application program; it is automatically handled by Natural. For further information, see *SELECT- Cursor-Oriented* below.
INTO	The `INTO` clause is used to specify the target fields in the program which are to be filled with the result of the selection. For further information and examples, see *INTO Clause* below.
VIEW	If one or more views are referenced in the `INTO` clause, the number of items specified in the *selection* must correspond to the number of fields defined in the view(s) (not counting group fields, redefining fields and indicator fields). For further information and examples, see *VIEW Clause* below.
table-expression	The *table-expression* consists of a `FROM` clause and an optional `WHERE` clause. For further information and examples, see *table-expression* below.
UNION	`UNION` unites the results of two or more *select-expressions*. For further information and an example, see *Query Involving UNION* below.
ORDER BY	The `ORDER BY` clause arranges the result of a `SELECT` statement in a particular sequence. For further information and examples, see *ORDER BY Clause* below.
IF NO RECORDS FOUND	The `IF NO RECORDS FOUND` clause is used to initiate a processing loop if no records meet the selection criteria specified in the preceding `SELECT` statement. For further information, see *IF NO RECORDS FOUND Clause* below.
END-SELECT	The Natural reserved keyword `END-SELECT` must be used to end the `SELECT` statement.

The following syntax elements belong to the SQL Extended Set:

WITH_CTE *common-table-expression,...*	**WITH_CTE *common-table-expression*:** This optional clause allows you to define a result table which can be referenced in any `FROM` clause of the `SELECT` that follows. Multiple common-table-expressions can be specified following the single `WITH_CTE` keyword. Each common-table-expression can also be referenced in the `FROM` clause of subsequent common-table-expression. For further information, see *SELECT- Cursor-Oriented*.
OPTIMIZE FOR	**OPTIMIZE FOR Clause:** For more information, see the *Natural for DB2* part in the *Database Management System Interfaces* documentation.

WITH CS/RS/UR/...	**WITH CS/RS/UR/...Clause:** This clause allows you to specify an explicit isolation level with which the statement is to be executed. For more information about this clause, see the section *Statement and System Variables* in the *Natural for DB2* of the *Database Management System Interfaces* documentation.
QUERYNO	**QUERYNO Clause:** The `QUERYNO` clause specifies the number to be used for this SQL statement in `EXPLAIN` output and trace records. For more information about this clause, see the section *Statement and System Variables* in the *Natural for DB2* part of the *Database Management System Interfaces* documentation.
FETCH FIRST	**FETCH FIRST Clause:** This clause limits the number of rows that can be fetched. For more information about this clause, see the section *Statement and System Variables* in the *Natural for DB2* part of the *Database Management System Interfaces* documentation.
WITH HOLD	**WITH HOLD Clause:** For more information, see the corresponding section in the *Natural for DB2* part of the *Database Management System Interfaces* documentation.
WITH RETURN	**WITH RETURN Clause:** For more information, see the corresponding section in the *Natural for DB2* part of the *Database Management System Interfaces* documentation.

WITH ... SCROLL	**WITH ... SCROLL Clause:** DB2 scrollable cursors are enabled with this clause. Scrollable cursors can be `ASENSITIVE`, `INSENSITIVE`, `SENSITIVE STATIC` or `SENSITIVE DYNAMIC`. ● `WITH ASENSITIVE SCROLL` specifies that the cursor is either `INSENSITIVE` or `SENSITIVE DYNAMIC`. This is determined by DB2 at open time of the cursor, depending on the read-only property of the cursor: If the cursor is read-only, the cursor will become `INSENSITIVE`. If the cursor is not read-only, the cursor will become `SENSITIVE DYNAMIC`. ● `WITH INSENSITIVE SCROLL` specifies that the cursor is insensitive for updates, deletes and inserts executed against the base table, after the cursor has been updated. Positioned updates and deletes are not allowed against `INSENSITIVE SCROLL` cursors. ● `WITH SENSITIVE STATIC` specifies that the cursor is sensitive for updates and deletes against the base table, but not against inserts, after the cursor has been opened. Positioned updates and deletes are allowed against `SENSITIVE STATIC SCROLL` cursors. ● `WITH SENSITIVE DYNAMIC` specifies that the cursor is sensitive for updates, deletes and inserts against the base table, after the cursor has been opened. Positioned updates and deletes are allowed against `SENSITIVE DYNAMIC SCROLL` cursors. Scrollable cursors allow the application to position any row in the cursor at any time as long as the cursor is open. The positioning is performed depending on the content of the *scroll_hv*. The content is evaluated each time a `FETCH` against DB2 is executed. For more information, see *SELECT - Cursor-Oriented*.

Syntax 2 - Non-Cursor Selection

Common Set Syntax

```
SELECT SINGLE

        selection INTO   { parameter ,...                                  } table-expression
                         { VIEW {view-name [correlation-name ]}, ...       }

        [IF-NO-RECORDS-FOUND-clause]
        statement...

{       END-SELECT (structured mode only) }
{       LOOP (reporting mode only)        }
```

Extended Set Syntax

```
SELECT SINGLE

   selection INTO        { parameter ,...                             } table-expression
                         { VIEW {view-name [correlation-name ]}, ...  }

   [   WITH    { CS }  ]
   [           { RR }  ]
   [           { UR }  ]

   [IF-NO-RECORDS-FOUND-clause]
   statement...

{ END-SELECT (structured mode only) }
{ LOOP (reporting mode only)        }
```

Syntax Element Description - Syntax 2:

SELECT SINGLE	The `SELECT SINGLE` statement supports the functionality of a non-cursor selection (singleton `SELECT`); that is, a select expression that retrieves at most one row without using a cursor. It cannot be referenced by a positioned `UPDATE` or a positioned `DELETE` statement. For further information, see *SELECT SINGLE - Non-Cursor-Oriented* in the *Natural for DB2* part of the *Database Management System Interfaces* documentation.
INTO	The `INTO` clause is used to specify the target fields in the program which are to be filled with the result of the selection. For further information and examples, see *INTO Clause* below.
VIEW	If one or more views are referenced in the `INTO` clause, the number of items specified in the *selection* must correspond to the number of fields defined in the view(s) (not counting group fields, redefining fields and indicator fields). For further information and examples, see *VIEW Clause* below.
table-expression	The *table-expression* consists of a `FROM` clause and an optional `WHERE` clause. For further information and examples, see *table-expression* below.
WITH CS/RR/UR	**WITH CS/RR/UR Clause:** This option allows you to specify an explicit isolation level with which the statement is to be executed. For more information about this clause, see the section *Statement and System Variables* in the *Natural for DB2* part of the *Database Management System Interfaces* documentation.
IF NO RECORDS FOUND	The `IF NO RECORDS FOUND` clause is used to initiate a processing loop if no records meet the selection criteria specified in the preceding `SELECT` statement. For further information, see *IF NO RECORDS FOUND Clause* below.
END-SELECT	The Natural reserved keyword `END-SELECT` must be used to end the `SELECT` statement.

INTO Clause

$$\texttt{INTO}\left\{\begin{array}{l}\textit{parameter}\ \texttt{,...}\\ \texttt{VIEW}\ \{\textit{view-name}\ [\textit{correlation-name}\ ]\}\texttt{,...}\end{array}\right\}$$

The `INTO` clause is used to specify the target fields in the program which are to be filled with the result of the selection. The `INTO` clause can specify either single *parameters* or one or more views as defined in the `DEFINE DATA` statement.

All target field values can come either from a single table or from more than one table as a result of a join operation (see also the section *Join Queries*).

Note:
In standard SQL syntax, an `INTO` clause is only used in non-cursor select operations (singleton `SELECT`) and can be specified only if a single row is to be selected. In Natural, however, the `INTO` clause is used

for both cursor-oriented and non-cursor select operations.

The *selection* can also merely consist of an asterisk (*). In a standard select expression, this is a shorthand for a list of all column names in the table(s) specified in the `FROM` clause. In the Natural `SELECT` statement, however, the same syntactical item `SELECT *` has a different semantic meaning: all the items listed in the `INTO` clause are also used in the selection. Their names must correspond to names of existing database columns.

Examples:

Example 1:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE
...
SELECT *
  INTO NAME, AGE
```

Example 2:

```
...
SELECT *
  INTO VIEW PERS
```

These examples are equivalent to the following ones:

Example 3:

```
...
SELECT NAME, AGE
  INTO NAME, AGE
```

Example 4:

```
...
SELECT NAME, AGE
  INTO VIEW PERS
```

VIEW Clause

`VIEW` {*view-name* [*correlation-name*]}, **...**

If one or more views are referenced in the `INTO` clause, the number of items specified in the *selection* must correspond to the number of fields defined in the view(s) (not counting group fields, redefining fields and indicator fields).

Note:
Both the Natural target fields and the table columns must be defined in a Natural DDM. Their names, however, can be different, since assignment is made according to their sequence.

Example of `INTO` Clause with View:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
END-DEFINE
...
SELECT FIRSTNAME, AGE
  INTO VIEW PERS
  FROM SQL-PERSONNEL
...
```

The target fields `NAME` and `AGE`, which are part of a Natural view, receive the contents of the table columns `FIRSTNAME` and `AGE`.

parameter	If single parameters are specified as target fields, their number and formats must correspond to the number and formats of the *columns* and/or *scalar-expressions* specified in the corresponding selection as described above (for details, see *Scalar Expressions*).
	
	Example:
	
	`DEFINE DATA LOCAL`
	`01 #NAME   (A20)`
	`01 #AGE    (I2)`
	`END-DEFINE`
	`...`
	`SELECT NAME, AGE`
	`  INTO #NAME, #AGE`
	`  FROM SQL-PERSONNEL`
	`...`
	
	The target fields `#NAME` and `#AGE`, which are Natural program variables, receive the contents of the table columns `NAME` and `AGE`.

correlation-name	If the `VIEW` clause is used within a `SELECT *` construction where multiple tables are to be joined, *correlation-names* are required if the specified view contains fields that reference columns which exist in more than one of these tables. In order to know which column to select, all these columns are qualified by the specified *correlation-name* at generation of the selection list. The *correlation-name* assigned to a view must correspond to one of the *correlation-names* used to qualify the tables to be joined. See also the section Join Queries.
	
	Example:
	
	`DEFINE DATA LOCAL`
	`01 PERS VIEW OF SQL-PERSONNEL`
	`  02 NAME`
	`  02 FIRST-NAME`
	`  02 AGE`
	`END-DEFINE`
	`...`
	`SELECT *`
	`  INTO VIEW PERS A`
	`  FROM SQL-PERSONNEL A, SQL-PERSONNEL B`
	`...`

table-expression

The *table-expression* consists of a `FROM` clause and an optional `WHERE` clause. The `GROUP BY` and `HAVING` clauses are not permitted.

Example 1:

```
DEFINE DATA LOCAL
01 #NAME      (A20)
01 #FIRSTNAME (A15)
01 #AGE       (I2)
...
END-DEFINE
...
SELECT NAME, FIRSTNAME, AGE
  INTO #NAME, #FIRSTNAME, #AGE
  FROM SQL-PERSONNEL
    WHERE NAME IS NOT NULL
      AND AGE > 20
...
  DISPLAY #NAME #FIRSTNAME #AGE
END-SELECT
...
END
```

Example 2:

```
DEFINE DATA LOCAL
01 #COUNT    (I4)
...
END-DEFINE
...
SELECT SINGLE COUNT(*) INTO #COUNT FROM SQL-PERSONNEL
...
```

See further information on *selection* and *table-expression*.

Query Involving UNION

Note:
In the following, the term "SELECT statement" is used as a synonym for the whole query-expression consisting of multiple select expressions concatenated with `UNION` operations.

`UNION` unites the results of two or more *select-expressions*. The columns specified in the individual *select-expressions* must be `UNION`-compatible; that is, matching in number, type and format.

Redundant duplicate rows are always eliminated from the result of a `UNION` unless the `UNION` operator explicitly includes the `ALL` qualifier. With `UNION`, however, there is no explicit `DISTINCT` option as an alternative to `ALL`.

Example:

```
DEFINE DATA LOCAL
01 PERS VIEW OF SQL-PERSONNEL
  02 NAME
  02 AGE
  02 ADDRESS (1:6)
END-DEFINE
```

```
...
SELECT NAME, AGE, ADDRESS
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE AGE > 55
UNION ALL
SELECT NAME, AGE, ADDRESS
  FROM SQL-EMPLOYEES
  WHERE PERSNR < 100
ORDER BY NAME
...
END-SELECT
...
```

In general, any number of *select-expressions* can be concatenated with `UNION`.

The `INTO` clause must be specified with the first *select-expression* only.

Any `ORDER BY` clause must appear after the final *select-expression*; the ordering columns must be identified by number, not by name.

ORDER BY Clause

$$\textbf{ORDER BY} \left\{ \left\{ \begin{array}{c} \textit{integer} \\ \textit{column-reference} \end{array} \right\} \left[\begin{array}{c} \underline{\textbf{ASC}} \\ \textbf{DESC} \end{array} \right] \right\} ,\ldots$$

The `ORDER BY` clause arranges the result of a `SELECT` statement in a particular sequence.

Each `ORDER BY` clause must specify a column of the result table. In most `ORDER BY` clauses a column can be identified either by *column-reference* (that is, by an optionally qualified column name) or by column number. In a query involving `UNION`, a column must be identified by column number. The column number is the ordinal left-to-right position of a column within the *selection*, which means it is an *integer* value. This feature makes it possible to order a result on the basis of a computed column which does not have a name.

Example:

```
DEFINE DATA LOCAL
1 #NAME          (A20)
1 #YEARS-TO-WORK (I2)
END-DEFINE
...
SELECT NAME , 65 - AGE
  INTO #NAME, #YEARS-TO-WORK
  FROM SQL-PERSONNEL
  ORDER BY 2
  ...
```

The order specified in the `ORDER BY` clause can be either ascending (`ASC`) or descending (`DESC`). `ASC` is the default.

Example:

```
DEFINE DATA LOCAL
1 PERS VIEW OF SQL-PERSONNEL
1 NAME
1 AGE
1 ADDRESS   (1:6)
END-DEFINE
...
SELECT NAME, AGE, ADDRESS
  INTO VIEW PERS
  FROM SQL-PERSONNEL
  WHERE AGE = 55
  ORDER BY NAME DESC
  ...
```

See further information on *integer* values and *column-reference*.

IF NO RECORDS FOUND-Clause

Note:
This clause actually does not belong to Natural SQL; it represents Natural functionality which has been made available to SQL loop processing.

Structured Mode Syntax

```
IF NO [RECORDS] [FOUND]
{   ENTER          }
{   statement...   }
END-NOREC
```

Reporting Mode Syntax

```
IF NO [RECORDS] [FOUND]
{ ENTER                     }
{ statement                 }
{ DO statement... DOEND     }
```

The `IF NO RECORDS FOUND` clause is used to initiate a processing loop if no records meet the selection criteria specified in the preceding `SELECT` statement.

If no records meet the specified selection criteria, the `IF NO RECORDS FOUND` clause causes the processing loop to be executed once with an "empty" record. If this is not desired, specify the statement `ESCAPE BOTTOM` within the `IF NO RECORDS FOUND` clause.

If one or more statements are specified with the `IF NO RECORDS FOUND` clause, the statements are executed immediately before the processing loop is entered. If no statements are to be executed before entering the loop, the keyword `ENTER` must be used.

Note:
If the result set of the `SELECT` statement consists of a single row of NULL values, the `IF NO RECORDS FOUND` clause is not executed. This could occur if the selection list consists solely of one of the aggregate functions `SUM`, `AVG`, `MIN` or `MAX` on columns, and the set on which these aggregate functions operate is empty. When you use these aggregate functions in the above-mentioned way, you should therefore check the values of the corresponding null-indicator fields instead of using an `IF NO RECORDS FOUND` clause.

Database Values

Unless other value assignments are made in the statements accompanying an `IF NO RECORDS FOUND` clause, Natural resets to empty all database fields which reference the file specified in the current loop.

Evaluation of System Functions

Natural system functions are evaluated once for the empty record that is created for processing as a result of the `IF NO RECORDS FOUND` clause.

Join Queries

A join is a query in which data is retrieved from more than one table. All the tables involved must be specified in the `FROM` clause.

Example:

```
DEFINE DATA LOCAL
1 #NAME     (A20)
1 #MONEY    (I4)
END-DEFINE
...
SELECT NAME, ACCOUNT
  INTO #NAME, #MONEY
  FROM  SQL-PERSONNEL P, SQL-FINANCE F
  WHERE P.PERSNR = F.PERSNR
    AND F.ACCOUNT > 10000
    ...
```

A join always forms the Cartesian product of the tables listed in the `FROM` clause and later eliminates from this Cartesian product table all the rows that do not satisfy the join condition specified in the `WHERE` clause.

Correlation-names can be used to save writing if table names are rather long. *Correlation-names* must be used when a column specified in the selection list exists in more than one of the tables to be joined in order to know which of the identically named columns to select.

SELECT - Cursor-Oriented

Like the Natural `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` (reporting mode) or `END-SELECT` statement. 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.

Below is information on:

- WITH_CTE common-table-expression,…
- OPTIMIZE FOR integer ROWS
- WITH - Isolation Level
- QUERYNO
- FETCH FIRST
- WITH HOLD
- WITH RETURN
- WITH INSENSITIVE/SENSITIVE

WITH_CTE *common-table-expression*,…

This clause permits to define result tables that can be referenced in any `FROM` clause of the `SELECT` statement that follows.

The Natural specific keyword `WITH_CTE` corresponds to the SQL keyword `WITH`. `WITH_CTE` will be translated into the SQL keyword `WITH` by the Natural compiler.

Each common-table-expression has to obey the following syntax:

```
[common-table-expression-name [(column-name,...)] AS (fullselect) ]
```

Syntax Description:

common-table-expression-name	Has to be an unqualified SQL identifier and must be different from any other common-table-expression-name specified in the same statement. Each *common-table-expression-name* can be specified in the `FROM` clause of any *common-table-expression-name* following or in the `FROM` clause of the `SELECT` statement following.
column-name	Has to be an unqualified SQL identifier and must be unique within one *common-table-expression-name*.
AS (*fullselect*)	The number of *column-names* must match the number of columns of the *fullselect*.

A common-table-expression can be used

- in place of a view to avoid creating the view;
- when the same result table needs to be shared in a `fullselect` ;
- when the result needs to be derived using recursion.

Queries using recursion are useful in applications such as bill of material.

Example:

```
WITH_CTE
 RPL (PART,SUBPART,QUANTITY) AS
 (SELECT ROOT.PART,ROOT.SUBPART,ROOT.QUANTITY
    FROM HGK-PARTLIST ROOT
   WHERE ROOT.PART ='01'
  UNION ALL
  SELECT CHILD.PART,CHILD.SUBPART,CHILD.QUANTITY
    FROM  RPL PARENT, HGK-PARTLIST CHILD
    WHERE PARENT.SUBPART = CHILD.PART
  )
SELECT DISTINCT PART,SUBPART,QUANTITY
  INTO VIEW V1
  FROM RPL
  ORDER BY PART,SUBPART,QUANTITY
END-SELECT
```

OPTIMIZE FOR *integer* ROWS

[**OPTIMIZE FOR** *integer* **ROWS**]

The `OPTIMIZE FOR` *`integer`* `ROWS` clause is used to inform DB2 in advance of the number (*integer*) of rows to be retrieved from the result table. Without this clause, DB2 assumes that all rows of the result table are to be retrieved and optimizes accordingly.

This optional clause is useful if you know how many rows are likely to be selected, because optimizing for *integer* rows can improve performance if the number of rows actually selected does not exceed the *integer* value (which can be in the range from 0 to 2147483647).

Example:

```
SELECT name INTO
#name FROM table WHERE AGE = 2 OPTIMIZE FOR 100 ROWS
```

WITH - Isolation Level

```
[ WITH { CS                      } ]
[        RR                      } ]
[        RR KEEP UPDATE LOCK     } ]
[        RS                      } ]
[        RS KEEP UPDATE LOCKS    } ]
[        UR                      } ]
```

This `WITH` clause allows you to specify an explicit isolation level with which the statement is to be executed. The following options are provided:

Option	**Meaning**
CS	Cursor Stability
RR	Repeatable Read
RS	Read Stability
RS KEEP UPDATE LOCKS	Only valid if a `FOR UPDATE OF` clause is specified. Read Stability and retaining update locks.
RR KEEP UPDATE LOCKS	Only valid if a `FOR UPDATE OF` clause is specified. Repeatable Read and retaining update locks.
UR	Uncommitted Read

`WITH UR` can only be specified within a `SELECT` statement and when the table is read-only. The default isolation level is determined by the isolation of the package or plan into which the statement is bound. The default isolation level also depends on whether the result table is read-only or not. To find out the default isolation level, refer to the IBM literature.

Note:
This option also works for non-cursor selection.

QUERYNO

[QUERYNO *integer***]**

The `QUERYNO` clause specifies the number to be used for this SQL statement in `EXPLAIN` output and trace records. The number is used as `QUERYNO` column in the `PLAN_TABLE` for the rows that contain information on this statement.

FETCH FIRST

```
[ FETCH FIRST { 1       } { ROWS } ONLY ]
              { integer } { ROW  }
```

The `FETCH FIRST` clause limits the number of rows to be fetched. It improves the performance of queries with potentially large result sets if only a limited number of rows is needed.

WITH HOLD

```
[WITH HOLD]
```

The `WITH HOLD` clause is used to prevent cursors from being closed by a commit operation within database loops. If `WITH HOLD` is specified, a commit operation commits all the modifications of the current logical unit of work, but releases only locks that are not required to maintain the cursor. This optional clause is mainly useful in batch mode; it is ignored in CICS pseudo-conversational mode and in IMS message-driven programs.

Example:

```
SELECT name INTO #name FROM table
WHERE AGE = 2 WITH HOLD
```

WITH RETURN

```
[WITH RETURN]
```

The `WITH RETURN` clause is used to create result sets. Therefore, this clause only applies to programs which operate as Natural stored procedure. If the `WITH RETURN` clause is specified in a `SELECT` statement, the underlying cursor remains open when the associated processing loop is left, except when the processing loop had read all rows of the result set itself. During first execution of the processing loop, only the cursor is opened. The first row is not yet fetched. This allows the Natural program to return a full result set to the caller of the stored procedure. It is up to you to decide how many rows are processed by the Natural stored procedure and how many unprocessed rows of the result set are returned to the caller of the stored procedure. If you want to process rows of the select operation in the Natural stored procedure, you must define

```
IF *counter =1 ESCAPE TOP END-IF
```

in order to avoid processing of the first "empty row" in the processing loop. If you decide to terminate the processing of rows, you must define

```
If condition ESCAPE BOTTOM END-IF
```

in the processing loop.

If the program reads all rows of the result set, the cursor is closed and no result set is returned for this `SELECT WITH RETURN` to the caller of the stored procedure.

The following programs are examples for retrieving full result sets (Example 1) and partial result sets (Example 2).

Example 1:

```
DEFINE DATA LOCAL
. . .
END DEFINE
*
*  Return all rows of the result set
*
SELECT * INTO VIEW V2
                    FROM SYSIBM-SYSROUTINES
                    WHERE RESULT_SETS > 0
                    WITH RETURN
ESCAPE BOTTOM
END-SELECT
END
```

Example 2:

```
DEFINE DATA LOCAL
. . .
END DEFINE
*
*  Read the first two rows and return the rest as result set
*
SELECT * INTO VIEW V2
                    FROM SYSIBM-SYSROUTINES
                    WHERE RESULT_SETS > 0
                    WITH   RETURN
WRITE PROCEDURE *COUNTER
IF *COUNTER = 1 ESCAPE TOP END-IF
IF *COUNTER = 3 ESCAPE BOTTOM END-IF
END-SELECT
END
```

WITH INSENSITIVE/SENSITIVE

```
[ WITH { ASENSITIVE SCROLL              } [:] scroll_hv  [GIVING  [:]  sqlcode] ]
       { INSENSITIVE SCROLL             }
         SENSITIVE STATIC SCROLL
         SENSITIVE DYNAMIC SCROLL
```

Natural for DB2 supports DB2 scrollable cursors by using the clauses `WITH ASENSITIVE SCROLL`, `WITH SENSITIVE STATIC SCROLL` and `SENSITVE DYNAMIC SCROLL`. Scrollable cursors allow Natural for DB2 applications to position randomly any row in a result set. With non-scrollable cursors, the data can only be read sequentially, from top to bottom.

`ASENSITIVE` scrollable cursors are either `INSENSITIVE` - if the cursor is `READ-ONLY` - or `SENSITIVE DYNAMIC` - if the cursor is not `READ-ONLY`.

`INSENSITIVE` and `SENSITIVE STATIC` scrollable cursors use temporary result tables and require a TEMP database in DB2 (see the relevant DB2 literature by IBM).

`INSENSITIVE SCROLL` refers to a cursor that cannot be used in Positioned `UPDATE` or Positioned `DELETE` operations. In addition, once opened, an `INSENSITIVE SCROLL` cursor does not reflect `UPDATE`s, `DELETE`s or `INSERT`s against the base table, after the cursor was opened.

`SENSITIVE STATIC SCROLL` refers to a cursor that can be used for Positioned `UPDATE`s or Positioned `DELETE` operations. In addition, a `SENSITIVE STATIC SCROLL` cursor reflects `UPDATE`s, `DELETE`s of base table rows. The cursor does not reflect `INSERT` operations.

`SENSITIVE DYNAMIC` scrollable cursors reflect `UPDATE`s, `DELETE`s and `INSERT`s against the base table while the cursor is open.

Below is information on:

- scroll_hv
- scroll_hv - Sensitivity Specification
- scroll_hv - Options
- GIVING [:] sqlcode

scroll_hv

The variable *scroll_hv* must be alphanumeric.

The variable *scroll_hv* specifies which row of the result table will be fetched during one execution of the database processing loop. Additionally, it specifies the sensitivity of `UPDATE`s or `DELETE`s against the base table row during a `FETCH` operation. The contents of *scroll_hv* is evaluated each time the database processing loop cycle is executed.

```
[ { INSENSITIVE } ]  { AFTER                                   }
[ { SENSITVE    } ]  { BEFORE                                  }
                     { CURRENT                                 }
                     { FIRST                                   }
                     { LAST                                    }
                     { PRIOR                                   }
                     { NEXT | N                                }
                     {                                         }
                     {                                         }
                     { { ABSOLUTE } [+ | - ] integer           }
                     { { RELATIVE }                            }
```

scroll_hv - Sensitivity Specification

The specification of the sensitivity `INSENSITIVE` or `SENSITIVE` is optional.

If it is omitted from a `FETCH` against an `INSENSITIVE SCROLL` cursor, the default will be `INSENSITIVE`.

If it is omitted from a `FETCH` against a `SENSITIVE STATIC/DYNAMIC SCROLL` cursor, the default will be `SENSITIVE`.

The sensitivity specifies whether or not the rows in the base table are checked when performing a `FETCH` operation for a scrollable cursor.

If the corresponding base table column qualifies for the `WHERE` clause and has not been deleted, a `SENSITIVE FETCH` will return the row of the base table.

If the corresponding base table column does not qualify for the `WHERE` clause or has not been deleted, a `SENSITIVE FETCH` will return an `UPDATE` hole or a `DELETE` hole state (SQLCODE +222).

An `INSENSITIVE FETCH` will not check the corresponding base table column.

scroll_hv - Options

Below is an explanation of the options available to determine the row(s) to fetch, the position from where to start the fetch and/or the direction in which to scroll:

Option	**Explanation**
AFTER	Positions after the last row. No row is fetched.
BEFORE	Positions before the first. No row is fetched.
CURRENT	Fetches the current row (again).
FIRST	Fetches the first row.
LAST	Fetches the last row.
NEXT	Fetches the row after the current one. This is the default value.
PRIOR	Fetch the row before the current one.
+/- integer	Only applies in connection with `ABSOLUTE` or `RELATIVE`. Specifies the position of the row to be fetched `ABSOLUTE` or `RELATIVE`. Enter a plus (+) or minus (-) sign followed by an integer. The default value is a plus (+).
ABSOLUTE	Only applies in connection with *+/- integer*. Uses *integer* as the absolute position within the result set from where the row is fetched. See the DB2 SQL reference by IBM about further details regarding positive and negative position numbers.
RELATIVE	Only applies in connection with *+/- integer*. Uses *integer* as the relative position to the current position within the result set from where the row is fetched. See the DB2 SQL reference by IBM about further details regarding positive and negative position numbers.

GIVING [:] *sqlcode*

The specification of `GIVING [:] sqlcode` is optional. If specified, the Natural variable `[:] sqlcode` must be of the Format I4. The values for this variable are returned from the DB2 SQLCODE of the underlying `FETCH` operation. This allows the application to react to different statuses encountered while the scrollable cursor is open. The most important status codes indicated by SQLCODE are listed in the following table:

SQLCODE	Explanation
0	`FETCH` operation successful, data returned except for `FETCH` with option `BEFORE` or `AFTER`.
+100	Row not found, cursor still open, no data returned.
+222	`UPDATE` or `DELETE` hole, cursor still open, no data returned. The corresponding row of the base table has been updated or deleted, so that the row no longer qualifies for the `WHERE` clause.
+231	Fetch operation with the option `CURRENT`, but cursor not positioned on any row, no data returned. This occurs if the previous `FETCH` returned SQLCODE +100.

If you specify `GIVING [:] sqlcode`, the application must react to the different statuses. If an SQLCODE +100 is entered five times successively and without terminal I/O, the NDB runtime will issue Natural Error NAT3296 in order to avoid application looping. The application can terminate the processing loop by executing an `ESCAPE` statement.

If you do not specify `GIVING [:] sqlcode`, except for SQLCODE 0 and SQLCODE +100, each SQLCODE will generate Natural Error NAT3700 and the processing loop will be terminated. SQLCODE +100 (row not found) will terminate the processing loop.

See also the example program DEM2SCRL supplied in the Natural system library SYSDB2.