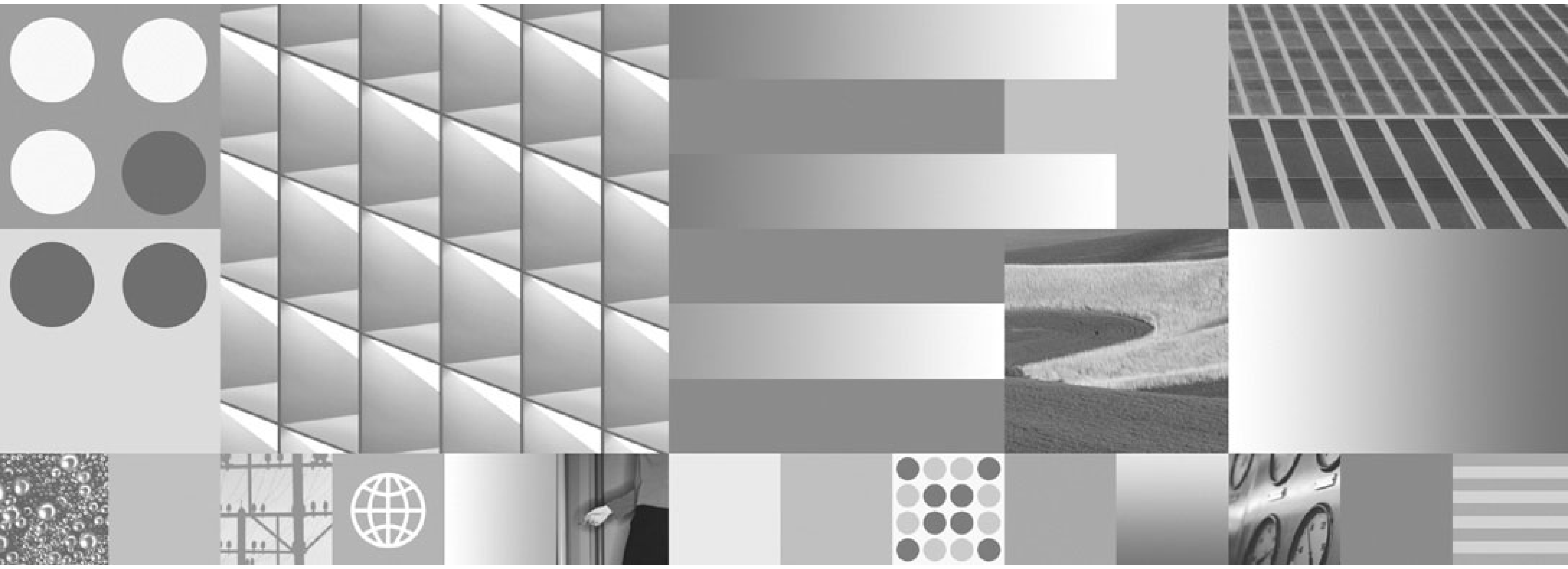
IBM DB2 9.7
for Linux, UNIX, and Windows

**Version 9 Release 7**

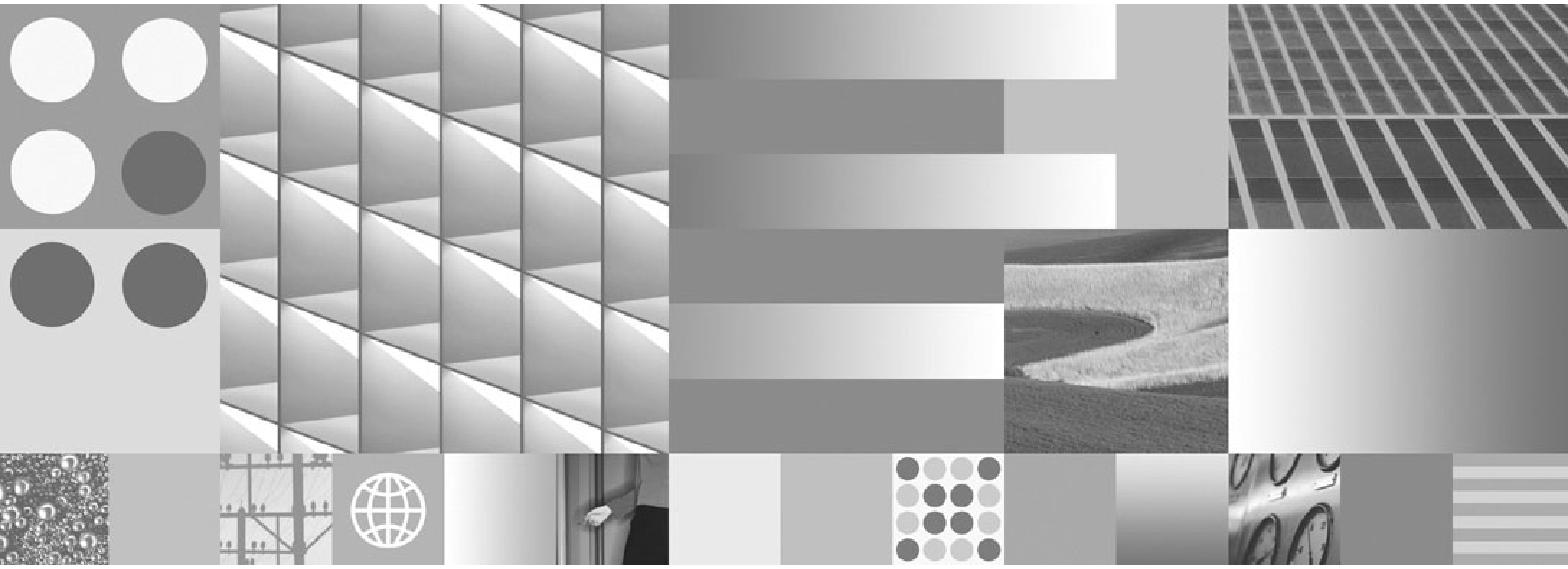


# SQL Procedural Languages: Application Enablement and Support
## Updated September, 2010

SC27-2470-02

IBM DB2 9.7
for Linux, UNIX, and Windows

**Version 9 Release 7**



## SQL Procedural Languages: Application Enablement and Support
## Updated September, 2010

SC27-2470-02

**Note**

Before using this information and the product it supports, read the general information under Appendix B, "Notices," on page 413.

**Edition Notice**

This document contains proprietary information of IBM. It is provided under a license agreement and is protected by copyright law. The information contained in this publication does not include any product warranties, and any statements provided in this manual should not be interpreted as such.

You can order IBM publications online or through your local IBM representative.

- To order publications online, go to the IBM Publications Center at www.ibm.com/shop/publications/order
- To find your local IBM representative, go to the IBM Directory of Worldwide Contacts at www.ibm.com/planetwide

To order DB2 publications from DB2 Marketing and Sales in the United States or Canada, call 1-800-IBM-4YOU (426-4968).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

**© Copyright IBM Corporation 1993, 2010.**
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

© Copyright IBM Corp. 1993, 2010

# Chapter 1. SQL Procedural Language (SQL PL)

The SQL Procedural Language (SQL PL) is a language extension of SQL that consists of statements and language elements that can be used to implement procedural logic in SQL statements. SQL PL provides statements for declaring variables and condition handlers, assigning values to variables, and for implementing procedural logic.

## Inline SQL PL

Inline SQL PL is a subset of SQL PL features that can be used in compound SQL (inlined) statements. Compound SQL (inlined) statements can be executed independently or can be used to implement the body of a trigger, SQL function, or SQL method. Compound SQL (inlined) statements can be executed independently from the DB2® CLP when it is in interactive mode to provide support for a basic SQL scripting language.

Inline SQL PL is described as "inline", because the logic is expanded into and executed with the SQL statements that reference them.

The following SQL PL statements are considered to be part of the set of inline SQL PL statements:

- Variable related statements
  - DECLARE <variable>
  - DECLARE <condition>
  - SET statement (assignment statement)
- Conditional statements
  - IF
  - CASE expression
- Looping statements
  - FOR
  - WHILE
- Transfer of control statements
  - GOTO
  - ITERATE
  - LEAVE
  - RETURN
- Error management statements
  - SIGNAL
  - GET DIAGNOSTICS

Other SQL PL statements that are supported in SQL procedures are not supported in compound SQL (inlined) statements. Cursors and condition handlers are not supported in inline SQL PL and therefore neither is the RESIGNAL statement.

Because inline SQL PL statements must be executed in compound SQL (inlined) statements, there is no support for PREPARE, EXECUTE, or EXECUTE IMMEDIATE statements.

© Copyright IBM Corp. 1993, 2010

Also, because ATOMIC must be specified in a compound SQL (inlined) statement that is dynamically prepared or executed, all or none of the member statements must commit successfully. Therefore the COMMIT and ROLLBACK statements are not supported either.

As for the LOOP and REPEAT statements, the WHILE statement can be used to implement equivalent logic.

Standalone scripting with inline SQL PL consists of executing a compound SQL (inlined) statement that is dynamically prepared or executed within a Command Line Processor (CLP) script or directly from a CLP prompt. Compound SQL (inlined) statements that are dynamically prepared or executed are bounded by the keywords BEGIN and END and must end with a non-default terminator character. They can contain SQL PL and other SQL statements.

Because inline SQL PL statements are expanded within the SQL statements that reference them rather than being individually compiled, there are some minor performance considerations that should be considered when you are planning on whether to implement your procedural logic in SQL PL in an SQL procedure or with inline SQL PL in a function, trigger, or compound SQL (compiled) statement that is dynamically prepared or executed.

## SQL PL in SQL procedures

SQL PL statements are primarily used in SQL procedures. SQL procedures can contain basic SQL statements for querying and modifying data, but they can also include SQL PL statements for implementing control flow logic around the other SQL statements. The complete set of SQL PL statements can be used in SQL procedures.

SQL procedures also support parameters, variables, assignment statements, a powerful condition and error handling mechanism, nested and recursive calls, transaction and savepoint support, and the ability to return multiple result sets to the procedure caller or a client application.

SQL PL, when used within SQL procedures, allows you to effectively program in SQL. The high-level language of SQL PL and the additional features that SQL procedures provide makes programming with SQL PL fast and easy to do.

As a simple example of SQL PL statements being used in a SQL procedure, consider the following example:

```
CREATE PROCEDURE UPDATE_SAL (IN empNum CHAR(6),
        INOUT rating SMALLINT)
LANGUAGE  SQL
BEGIN
  IF rating = 1 THEN
   UPDATE employee
      SET salary = salary * 1.10, bonus = 1000
        WHERE empno = empNum;
  ELSEIF rating = 2 THEN
    UPDATE employee
      SET salary = salary * 1.05, bonus = 500
        WHERE empno = empNum;
  ELSE
    UPDATE employee
      SET salary = salary * 1.03, bonus = 0
        WHERE empno = empNum;
  END IF;
END
```

## Inline SQL PL and SQL functions, triggers, and compound SQL statements

Inline SQL PL statements can be executed in compound SQL (compiled) statements, compound SQL (inlined) statements, SQL functions, and triggers.

A compound SQL (inlined) statement is one that allows you to group multiple SQL statements into an optionally atomic block in which you can declare variables, and condition handling elements. These statements are compiled by DB2 as a single SQL statement and can contain inline SQL PL statements.

The bodies of SQL functions and triggers can contain compound SQL (inlined) statements and can also include some inline SQL PL statements.

On their own, compound SQL (inlined) statements are useful for creating short scripts that perform small units of logical work with minimal control flow, but that have significant data flow. Within functions and triggers, they allow for more complex logic to be executed when those objects are used.

As an example of a compound SQL (inlined) statement that contains SQL PL, consider the following:

```
BEGIN ATOMIC
  FOR row AS
    SELECT pk, c1, discretize(c1) AS v FROM source
  DO
    IF row.v is NULL THEN
      INSERT INTO except VALUES(row.pk, row.c1);
    ELSE
      INSERT INTO target VALUES(row.pk, row.d);
    END IF;
  END FOR;
END
```

The compound SQL (inlined) statement is bounded by the keywords BEGIN and END. It includes use of both the FOR and IF/ELSE control-statements that are part of SQL PL. The FOR statement is used to iterate through a defined set of rows. For each row a column's value is checked and conditionally, based on the value, a set of values is inserted into another table.

As an example of a trigger that contains SQL PL, consider the following:

```
CREATE TRIGGER validate_sched
NO CASCADE BEFORE INSERT ON c1_sched
FOR EACH ROW
MODE DB2SQL
Vs: BEGIN ATOMIC

  IF (n.ending IS NULL) THEN
    SET n.ending = n.starting + 1 HOUR;
  END IF;

  IF (n.ending > '21:00') THEN
    SIGNAL SQLSTATE '80000'  SET MESSAGE_TEXT =
           'Class ending time is after 9 PM';
  ELSE IF (n.DAY=1 or n.DAY-7) THEN
    SIGNAL SQLSTATE '80001' SET MESSAGE_TEXT =
           'Class cannot be scheduled on a weekend';
  END IF;
END vs;
```

This trigger is activated upon an insert to a table named c1_sched and uses SQL PL to check for and provide a class end time if one has not been provided and to raise an error if the class end time is after 9 pm or if the class is scheduled on a weekend. As an example of a scalar SQL function that contains SQL PL, consider the following:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
RETURNS  DECIMAL(10,3)
LANGUAGE SQL  MODIFIES SQL
BEGIN
  DECLARE price DECIMAL(10,3);

  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table WHERE Pid = GetPrice.Pid);
  END IF;

  RETURN price;
END
```

This simple function returns a scalar price value, based on the value of an input parameter that identifies a vendor. It also uses the IF statement.

For more complex logic that requires output parameters, the passing of result sets or other more advanced procedural elements SQL procedures might be more appropriate.

# SQL PL data types

## Anchored data type

An anchored data type is a data type that is defined to be the same as that of another object. If the underlying object data type changes, the anchored data type also changes.

The following topics provide more information about anchored data types:

### Features of the anchored data type

An anchored type defines a data type based on another SQL object such as a column, global variable, SQL variable, SQL parameter, or the row of a table or view.

A data type defined using an anchored type definition maintains a dependency on the object to which it is anchored. Any change in the data type of the anchor object will impact the anchored data type. If anchored to the row of a table or view, the anchored data type is ROW with the fields defined by the columns of the anchor table or anchor view.

This data type is useful when declaring variables in cases where you require that the variable have the same data type as another object, for example a column in a table, but you do not know exactly what is the data type.

An anchored data type can be of the same type as one of:

- a row in a table
- a row in a view
- a cursor variable row definition
- a column in a table

- a column in a view
- a local variable, including a local cursor variable or row variable
- a global variable

Anchored data types can only be specified when declaring or creating one of the following:

- a local variable in an SQL procedure, including a row variable
- a local variable in a compiled SQL function, including a row variable
- a routine parameter
- a user-defined cursor data type using the CREATE TYPE statement.
  - It cannot be referenced in a DECLARE CURSOR statement.
- a function return data type
- a global variable

To define an anchored data type specify the ANCHOR DATA TYPE TO clause (or the shorter form ANCHOR clause) to specify what the data type will be. If the anchored data type is a row data type, the ANCHOR ROW OF clause, or one of its synonyms, must be specified. These clauses are supported within the following statements:

- DECLARE
- CREATE TYPE
- CREATE VARIABLE
  - In this version, global variables can only be anchored to other global variables, a column in a table, or a column in a view.

## Restrictions on the anchored data type

Review the restrictions on the use of the anchored data type before declaring variables of this type or when troubleshooting problems related to their use.

The following restrictions apply to the use of anchored data types, including types specified using the PL/SQL %TYPE attribute:

- Anchored data types are not supported in inline SQL functions.
- Anchored data types cannot reference nicknames or columns in nicknames.
- Anchored data types cannot reference typed tables, columns of typed tables, typed views, or columns of typed views.
- Anchored data types cannot reference declared temporary tables, or columns of declared temporary tables.
- Anchored data types cannot reference row definitions associated with a weakly typed cursor.
- Anchored data types cannot reference objects with a code page or collation that is different from the database code page or database collation.

## Anchored data type variables

An anchored variable is a local variable or parameter with a data type that is an anchored data type.

Anchored variables are supported in the following contexts:

- SQL procedures
  - In SQL procedures, parameters and local variables can be specified to be of an anchored data type.
- Compiled SQL functions

  - SQL functions created using the CREATE FUNCTION statement that specify the BEGIN clause instead of the BEGIN ATOMIC clause can include parameter or local variable specification that are of the anchored data type.
- Module variables
  - Anchored variables can be specified as published or unpublished variables defined within a module.
- Global variables
  - Global variables can be created of the anchored data type.

Anchored variables are declared using the DECLARE statement.

## Declaring local variables of the anchored data type

Declaring local variables or parameters of the anchored data type is a task that you would perform whenever it is necessary that the data type of the variable or parameter remain consistent with the data type of the object to which it is anchored.

The object of the data type that the variable will be anchored to must be defined.

1. Formulate a DECLARE statement
   a. Specify the name of the variable.
   b. Specify the ANCHOR DATA TYPE TO clause.
   c. Specify the name of the object that is of the data type that the variable is to be anchored.
2. Execute the DECLARE statement from a supported DB2 interface.

If the DECLARE statement executes successfully, the variable is defined in the database with the specified anchor data type.

The following is an example of an anchored data type declaration in which a variable named v1 is anchored to the data type of a column named c1 in a table named emp:

```
DECLARE v1 ANCHOR DATA TYPE TO emp.c1;
```

Once the variable is defined it can be assigned a value, be referenced, or passed as a parameter to routines.

## Examples: Anchored data type use

Examples of anchored data type use can be useful as a reference when using this data type.

The following topics include examples of anchored data type use:

**Example: Variable declarations of anchored data types:**

Examples of anchored data type declarations can be useful references when declaring variables.

The following is an example of a declaration of a variable named v1 that has the same data type as the column name in table staff:

```
DECLARE v1 ANCHOR DATA TYPE TO staff.name;
```

The following is an example of a CREATE TYPE statement that defines a type named empRow1 that is anchored to a row defined in a table named employee:

```
CREATE TYPE empRow1 AS ROW ANCHOR DATA TYPE TO ROW OF employee;
```

For variables declared of type empRow1, the field names are the same as the table column names.

If the data type of the column name is VARCHAR(128), then the variable v1 will also be of data type VARCHAR(128).

**Examples: Anchored data type use in SQL routines:**

Examples of anchored data type use in SQL routines are useful to reference when creating your own SQL routines.

The set of examples below demonstrate various features and uses of anchored data types in SQL routines. The anchored data type features are demonstrated more so than the features of the SQL routines that contain them.

The following is an example that demonstrates a declared variable that is anchored to the data type of a column in a table:

```
CREATE TABLE tab1(col1 INT, col2 CHAR)@

INSERT INTO tab1 VALUES (1,2)@

INSERT INTO tab1 VALUES (3,4)@

CREATE TABLE tab2 (col1a INT, col2a CHAR)@

CREATE PROCEDURE p1()
BEGIN
  DECLARE var1 ANCHOR tab1.col1;
  SELECT col1 INTO var1 FROM tab1 WHERE col2 = 2;
  INSERT INTO tab2 VALUES (var1, 'a');
END@

CALL p1()@
```

When the procedure p1 is called, the value of the column col1 for a particular row is selected into the variable var1 of the same type.

The following CLP script includes an example of a function that demonstrates the use of an anchored data type for a parameter to a function:

```
-- Create a table with multiple columns
CREATE TABLE tt1 (c1 VARCHAR(18), c2 CHAR(8), c3 INT, c4 FLOAT)
@

INSERT INTO tt1 VALUES ('aaabbb', 'ab', 1, 1.1)
@

INSERT INTO tt1 VALUES ('cccddd', 'cd', 2, 2.2)
@

SELECT c1, c2, c3, c4 FROM tt1
@

-- Creation of the function
CREATE FUNCTION func_a(p1 ANCHOR tt1.c3)
 RETURNS INT
 BEGIN
   RETURN p1 + 1;
 END
@

-- Invocation of the function
SELECT c1, c2 FROM tt1 WHERE c3 = func_a(2)
```

```
@

-- Another invocation of the function
SELECT c1, c2 FROM tt1 WHERE c3 = func_a(1)
@

DROP FUNCTION func_a
@

DROP TABLE tt1
@
```

When the function func_a is invoked, the function performs a basic operation using the value of the anchored data type parameter.

## Row types

A row data type is a user-defined type containing an ordered sequence of named fields each with an associated data type.

A row type can be used as the type for global variables, SQL variables, and SQL parameters in SQL PL to provide flexible manipulation of the columns in a row of data, typically retrieved using a query.

### Features of the row data type

The features of the row data type make it useful for simplifying SQL PL code.

The row data type is supported for use with the SQL Procedural language only. It is a structure composed of multiple fields each with their own name and data type that can be used to store the column values of a row in a result set or other similarly formatted data.

This data type can be used to:

- Simplify the coding of logic within SQL Procedural Language applications. For example, database applications process records one at a time and require parameters and variables to temporarily store records. A single row data type can replace the multiple parameters and variables required to otherwise process and store the record values. This greatly simplifies the ability to pass row values as parameters within applications and routines.
- Facilitate the porting to DB2 SQL PL of code written in other procedural SQL languages that support a similar data type.
- Reference row data in data-change statements and queries including: INSERT statement, FETCH statement, VALUES INTO statement and SELECT INTO statement.

Row data types must be created using the CREATE TYPE (ROW) statement. Once created, variables of the defined data type can be declared within SQL PL contexts using the DECLARE statements. These variables can then be used to store values of the row type.

Row field values can be explicitly assigned and referenced using single-dot, "." notation.

### Restrictions on the row data type

It is important to note the restrictions on the use of the row data type before using it or when troubleshooting an error that might be related to its use.

The following restrictions apply to the row data type:

- The maximum number of fields supported in a row data type is 1012.
- The row data type cannot be passed as an input parameter value to procedures and functions from the CLP.
- The row data type cannot be passed as an input-output or output parameter value from procedures and functions to the CLP.
- Row data type variables cannot be directly compared. To compare row type variables, each field can be compared.
- The following data types are not supported for row fields:
  - XML data type
  - LONG VARCHAR
  - LONG VARGRAPHIC
  - structured data types
  - row data types
  - array data types
- Global variables of type row that contain one or more fields of type LOB are not supported.
- Use of the CAST function to cast a parameter value to a row data type is not supported.

Other general restrictions might apply related to the use of a data type, authorizations, execution of SQL, scope of use of the data type or other causes.

## Row variables

Row variables are variables based on user-defined row data types. Row variables can be declared, assigned a value, set to another value, or passed as a parameter to and from SQL procedures. Row variables inherit the properties of the row data types upon which they are based. Row variables are used to hold a row of data from within a result set or can be assigned other tuple-format data.

Row variables can be declared within SQL procedures using the DECLARE statement.

## Creating row variables

To create row variables you must first create the row type and then declare the row variable.

The following topics show you how to create the row data type and variable:

**Creating a row data type:**

Creating a row data type is a prerequisite to creating a row variable.

Before you create a row data type:
- Read: “Row types” on page 8
- Read: “Restrictions on the row data type” on page 8

This task can be done from any interface that supports the execution of the CREATE TYPE statement.

To create a row data type within a database, you must successfully execute the CREATE TYPE (ROW) statement from any DB2 interface that supports the execution of SQL statements.

1. Formulate a CREATE TYPE (ROW) statement:
   a. Specify a name for the type.
   b. Specify the row field definition for the row by specifying a name and data type for each field in the row.

   The following is an example of how to create a row data type that can be associated with result sets with the same format as the empRow row data type:

   ```
   CREATE TYPE empRow AS ROW (name VARCHAR(128), id VARCHAR(8));
   ```
2. Execute the CREATE TYPE statement from a supported DB2 interface.

If the CREATE TYPE statement executes successfully, the row data type is created in the database. If the statement does not execute successfully, verify the syntax of the statement and verify that the data type does not already exist.

Once the row data type is created, row variables can be declared based on this data type.

**Declaring local variables of type row:**

Variables of type row can be declared once the row data type has been created.

Before you create a row data type:
- Read: "Row types" on page 8
- Read: "Restrictions on the row data type" on page 8

Row data type variables can only be declared in SQL PL contexts including SQL procedures and functions where execution of the DECLARE statement is supported.

The following steps must be followed to declare a local row variable:
1. Formulate a declare statement:
   a. Specify a name for the variable.
   b. Specify the row data type that will define the variable. The specified row data type must be already defined in the database.

   The following is an example of how to formulate a DECLARE statement that defines a row variable of type empRow:

   ```
   DECLARE r1 empRow;
   ```
2. Execute the DECLARE statement within a supported context.

If execution of the DECLARE statement is successful, the row variable is created.

Upon creation of a row variable each field in the row is initialized to a NULL value.

The row variable can be assigned values, be referenced, or passed as a parameter.

## Assigning values to row variables

Values can be assigned to variables of type row in multiple ways. A row variable value can be assigned to another row variable. Variable field values can be assigned and referenced. Field values of a row are referenced by using a single-dot "." notation.

The following topics show how to assign values to row type variables and arrays of row type variables:

**Supported assignments to row data types:**

A variety of values are supported for assignment to rows and row fields.

When a row variable or parameter is declared, each field in the row has a default value of NULL until a value is assigned to it.

The following types of values can be assigned to a row variable:

- another row variable of the same row data type using the SET statement
  - Row variable values can only be assigned to row variables if they are type compatible. Two row variables are compatible if they are both of the same row data type or if the source row variable is anchored to a table or view definition. For two variables to be type compatible, it is not sufficient for them to have the same field names and field data types.

    For example, if a row data type named row1 is created and another data type named row2 is created and they are identical in definition, the value of a variable of type row1 cannot be assigned to the variable of type row2. Nor can the value of the variable of type row2 be assigned to the variable of type row1. However, the value of variable v1 of type row1 can be assigned to a variable v2 that is also of type row1.
- A tuple with the same number of elements as the row and elements of the same data types as the fields of the row.
  - The following is an example of a literal tuple being assigned to a row:

    ```
    SET v1 = (1, 'abc')
    ```
- expression that resolves to a row value
  - An example of an expression that resolves to a row value that can be assigned to a row variable is the resolved expression in a VALUES ... INTO statement. The following is an example of such an assignment:

    ```
    VALUES (1, 'abc') INTO rv1
    ```
- the return type of a function (if it is of the same row data type as the target variable):
  - The following is an example where the return type of a function named foo is of the same row data type as the target variable:

    ```
    SET v1 = foo()
    ```

    If the return data type is defined as an anchored data type, the anchored data type assignment rules apply.
- the single row result set of a query
  - The result set must have the same number of elements as the row and the columns must be assignable to the same data types as the fields of the row. The following is an example of this type of assignment:

    ```
    SET v1 = (select c1, c2 from T)
    ```
- NULL
  - When NULL is assigned to a row variable, all the row fields are set to NULL but the row variable itself remains NOT NULL.

The following types of values can be assigned to a row variable field:

- literal
- parameter
- variable
- expression

- NULL

Values can be assigned to row field values in the following ways:

- Using the SET statement
- Using a SELECT INTO statement that resolves to a row value
- Using a FETCH INTO statement that resolves to a row value
- Using a VALUES INTO statement that resolves to a row value

The ROW data type can be specified as the return-type of an SQL scalar function.

**Assigning values to a row variable using the SET statement:**

Assigning values to a row variable can be done using the SET statement. A row value can be assigned to a row variable. A row field value or expression can be assigned to a row field.

Row values can be assigned to row variables using the SET statement if they are both of the same user-defined row data type.

The following is an example of how to assign a row value to a row variable of the same format:

```
SET empRow = newHire;
```

The row value newHire has the same format as the empRow variable - the number and types of the row fields are identical:

```
empRow.lastName      /* VARCHAR(128) */
empRow.firstName     /* VARCHAR(128) */
empRow.id            /* VARCHAR(10)  */
empRow.hireDate      /* TIMESTAMP    */
empRow.dept          /* VARCHAR(3)   */

newHire.lastName     /* VARCHAR(128) */
newHire.firstName    /* VARCHAR(128) */
newHire.id           /* VARCHAR(10)  */
newHire.hireDate     /* TIMESTAMP    */
newHire.dept         /* VARCHAR(3)   */
```

If you attempt to assign a row value to a variable that does not have an identical format an error will be raised.

Row values can be assigned by assigning values to the individual fields in a row. The following is an example of how to assign values to the fields in the row named empRow using the SET statement:

```
SET empRow.lastName = 'Brown';           // Literal value assignment

SET empRow.firstName = parmFirstName;    // Parameter value of same type assignment

SET empRow.id = var1;                    // Local variable of same type assignment

SET empRow.hiredate = CURRENT_TIMESTAMP; // Special register expression assignment

SET empRow.dept = NULL;                  // NULL value assignment
```

Any supported field assignment can be used to initialize a row value.

**Assigning row values to row variables using SELECT, VALUES, or FETCH statements:**

A row value can be assigned to a variable of type row by using a SELECT INTO statement, a VALUES INTO statement, or a FETCH INTO statement. The field values of the source row value must be assignable to the field values of the target row variable.

The following is an example of how to assign a single row value to a row variable named empRow using a SELECT statement:

```
SELECT * FROM employee
INTO empRow
WHERE id=5;
```

If the select query resolves to more than one row value, an error is raised.

The following is an example of how to assign a row value to a row variable named empEmpBasics using a VALUES INTO statement:

```
VALUES (5, 'Jane Doe', 10000) INTO empBasics;
```

The following is an example of how to assign a row value to a row variable named empRow using a FETCH statement that references a cursor named cur1 that defines a row with compatible field values as the variable empRow:

```
FETCH cur1 INTO empRow;
```

Other variations of use are possible with each of these statements.

## Comparing row variables and row field values

Row variables cannot be directly compared even if they are the same row data type, however individual row fields can be compared.

Individual fields within a row type can be compared to other values and the comparison rules for the data type of the field apply.

To compare two row variables, individual corresponding field values must be compared.

The following is an example of a comparison of two row values with compatible field definitions in SQL PL:

```
  IF ROW1.field1 = ROW2.field1 AND
     ROW1.field2 = ROW2.field2 AND
     ROW1.field3 = ROW2.field3
  THEN
     SET EQUAL = 1;
  ELSE
     SET EQUAL = 0;
```

In the example the IF statement is used to perform the procedural logic which sets a local variable EQUAL to 1 if the field values are equal, or 0 if the field values are not equal.

## Referencing row values

Row values and row field values can be referenced within SQL and SQL statements.

The following topics demonstrate where and how row values can be referenced:

**Referencing row variables:**

Row variable values can be referenced by name wherever row variable data type references are supported.

Supported row variable reference contexts include the following:

- Source or target of a SET statement
- INSERT statement
- Target of SELECT INTO, VALUES INTO, or FETCH statements

The following is an example of a row variable being assigned to another row variable with the same definition using the SET statement:

```
-- Referencing row variables as source and
   target of a SET statement
SET v1 = v2;
```

The following is an example of row variables being referenced in an INSERT statement that inserts two rows. The row variables v1 and v2 have a field definition that is type compatible with the column definition of the table that is the target of the INSERT statement:

```
-- Referencing row variables in an INSERT statement
INSERT INTO employee VALUES v1, v2;
```

The following is an example of a row variable being referenced in a FETCH statement. The row variable empRow has the same column definition as the result set associated with the cursor c1:

```
-- Referencing row variables in a FETCH statement
FETCH c1 INTO empRow;
```

The following is an example of a row variable named v3 being referenced in a SELECT statement. Two column values in the employee table are being selected into the two fields of the variable v3:

```
-- Referencing row variables in a SELECT statement
SELECT id, name INTO v3 FROM employee;
```

**Referencing fields in row variables:**

Field values can be referenced in multiple contexts.

A row field value can be referenced wherever a value of the field's data type is permitted. The following contexts are supported for referencing row fields:

- Wherever a value of the field's data type is permitted including, but not limited to:
  - As the source of an assignment (SET statement)
  - As the target of an assignment (SET statement)
  - As the target of SELECT INTO, VALUES INTO, or FETCH INTO statement.

To reference the values of fields in a row variable a single-dot notation is used. Field values are associated with variables as follows:

```
<row-variable-name>.<field-name>
```

The following is an example of how to access the field id of variable employee:

```
employee.id
```

Examples of supported references to row variable field values follow.

The following is an example that shows how to assign a literal value to a field in row variable v1:

```
-- Literal assignment to a row variable field
SET v1.c1 = 5;
```

The following example shows how to assign literal and expression values to multiple row variable fields:

```
-- Literal assignment to fields of row variable
SET (emp.id, emp.name) = (v1.c1 + 1, 'James');
```

The following example shows how to reference field values in an INSERT statement:

```
-- Field references in an INSERT statement
INSERT INTO employee
VALUES(v1.c1, 'Beth'),
      (emp.id, emp.name);
```

The following example shows how to reference field values in an UPDATE statement:

```
-- Field references in an UPDATE statement
UPDATE employee
SET name = 'Susan'
WHERE id = v1.c1;
```

The following example shows how to reference field values in a SELECT INTO statement:

```
-- Field references in a SELECT INTO statement
SELECT employee.firstname INTO v2.c1
FROM employee
WHERE name=emp.name;
```

**Referencing row variables in INSERT statements:**

Row variables can be used in INSERT statements to append or modify an entire table row.

The following is an example of an INSERT statement that inserts a row into tabled employee:

```
INSERT INTO employee VALUES empRow;
```

For the INSERT statement, the number of fields in the row variable must match the number of columns in the implicit or explicit target column list.

The INSERT statement shown above inserts into each column in the table, the corresponding row field value. Thus the INSERT statement above is equivalent to the following INSERT statement:

```
INSERT INTO employee VALUES (emp.id,
                             emp.name,
                             emp.salary,
                             emp.phone);
```

## Passing rows as routine parameters

Row type values and arrays of row type variables can be passed as parameters to procedures and functions. Procedures support these data types as IN, OUT, and INOUT parameters.

The following is an example of a procedure that takes a CHAR type as an input parameter, modifies a field in the output row parameter and then returns.

```
CREATE PROCEDURE p(IN basicChar CHAR, OUT outEmpRow empRow)
BEGIN

  SET outEmpRow.field2 = basicChar;

END@
```

The following is an example of a CALL statement that invokes the procedure:

```
CALL p('1', myEmpRow)@
```

## Dropping a row data type

Dropping a row data type is done when the row data type is no longer required or when you want to reuse the name of an existing row data type.

The following prerequisites must be met before you can drop a row data type:

- A connection to the database must be established.
- The row data type must exist in the database.

Dropping a row data type is done when the row data type is no longer required or when you want to reuse the name of an existing row data type. Dropping a row can be done from any interface that supports the execution of the DROP statement.

1. Formulate a DROP statement that specifies the name of the row data type to be dropped.
2. Execute the DROP statement from a supported DB2 interface.

The following is an example of how to drop a row data type named simpleRow.

```
DROP TYPE simpleRow;
```

If the DROP statement executes successfully, the row data type is dropped from the database.

## Examples: Row data type use

Examples of row data type use provide a useful reference for understanding how and when to use the row data type.

The following topics demonstrate how to use the row data type:

**Example: Row data type use in a CLP script:**

Some basic features of row data types are shown within a DB2 CLP script to demonstrate how row data types are most commonly used.

The following DB2 CLP script demonstrates the use of the row data type and its related operations. It includes demonstrations of:

- Creating row data types
- Creating a table
- Creating a procedure that includes:
  - Row data type declarations
  - Inserting values to a type that include some row field values
  - Updating row values based on a row field value
  - Selecting values into a row field value
  - Assigning a row value to a row

  - Assigning row field values to a parameter
- Calling the procedure
- Dropping the row data types and table

```
-- Creating row types

CREATE TYPE row01 AS ROW (c1 INTEGER)@

CREATE TYPE empRow AS ROW (id INTEGER, name VARCHAR(10))@

CREATE TABLE employee (id INTEGER, name VARCHAR(10))@

CREATE procedure proc01 (OUT p0 INTEGER, OUT p1 INTEGER)
BEGIN
  DECLARE v1, v2 row01;
  DECLARE emp empRow;

  -- Assigning values to row fields
  SET v1.c1 = 5;
  SET (emp.id, emp.name) = (v1.c1 + 1, 'James');

  -- Using row fields in DML
  INSERT INTO employee
  VALUES (v1.c1, 'Beth'), (emp.id, emp.name);

  UPDATE employee
  SET name = 'Susan' where id = v1.c1;

  -- SELECT INTO a row field
  SELECT id INTO v2.c1
  FROM employee
  WHERE name = emp.name;

  -- Row level assignment
  SET v1 = v2;

  -- Assignment to parameters
  SET (p0, p1) = (v1.c1, emp.id);

END@

CALL proc01(?, ?)@

SELECT * FROM employee@

DROP procedure proc01@

DROP TABLE employee@

-- Dropping row types
DROP TYPE empRow@

DROP TYPE row01@
```

This script can be saved and run from a DB2 Command Line by issuing the following:

```
DB2 -td@ -vf <filename>;
```

The following is the output of running the script:

```
CREATE TYPE row01 AS ROW (c1 INTEGER)
DB20000I  The SQL command completed successfully.

CREATE TYPE empRow AS ROW (id INTEGER, name VARCHAR(10))
DB20000I  The SQL command completed successfully.
```

```
CREATE TABLE employee (id INTEGER, name VARCHAR(10))
DB20000I  The SQL command completed successfully.

CREATE procedure proc01 (OUT p0 INTEGER, OUT p1 INTEGER)
   BEGIN DECLARE v1, v2 row01;
   DECLARE emp empRow;
   SET v1.c1 = 5;
   SET (emp.id, emp.name) = (v1.c1 + 1, 'James');
   INSERT INTO employee  VALUES (v1.c1, 'Beth'), (emp.id, emp.name);
   UPDATE employee  SET name = 'Susan' where id = v1.c1;
   SELECT id INTO v2.c1 FROM employee WHERE name = emp.name;
   SET v1 = v2;
   SET (p0, p1) = (v1.c1, emp.id);
END

DB20000I  The SQL command completed successfully.

CALL proc01(?, ?)

  Value of output parameters
  --------------------------
  Parameter Name  : P0
  Parameter Value : 6

  Parameter Name  : P1
  Parameter Value : 6

  Return Status = 0

SELECT * FROM employee

ID          NAME
----------- ----------
          5 Susan
          6 James

  2 record(s) selected.


DROP procedure proc01
DB20000I  The SQL command completed successfully.

DROP TABLE employee
DB20000I  The SQL command completed successfully.

DROP TYPE empRow
DB20000I  The SQL command completed successfully.

DROP TYPE row01
DB20000I  The SQL command completed successfully.
```

**Example: Row data type use in an SQL procedure:**

The row data type can be used in SQL procedures to retrieve record data and pass it as a parameter.

This topic contains an example of a CLP script that includes the definitions of multiple SQL procedures that demonstrate some of the many uses of rows.

The procedure named ADD_EMP takes a row data type as an input parameter which it then inserts into a table.

The procedure named NEW_HIRE uses a SET statement to assign values to a row variable and passes a row data type value as a parameter in a CALL statement that invokes another procedure.

The procedure named FIRE_EMP selects a row of table data into a row variable and inserts row field values into a table.

The following is the CLP script - it is followed by the output of running the script from the CLP in verbose mode:

```
--#SET TERMINATOR @;
CREATE TABLE employee (id INT,
                       name VARCHAR(10),
                       salary DECIMAL(9,2))@

INSERT INTO employee VALUES (1, 'Mike', 35000),
                            (2, 'Susan', 35000)@

CREATE TABLE former_employee (id INT, name VARCHAR(10))@

CREATE TYPE empRow AS ROW ANCHOR ROW OF employee@
CREATE PROCEDURE ADD_EMP (IN newEmp empRow)
BEGIN
  INSERT INTO employee VALUES newEmp;
END@

CREATE PROCEDURE NEW_HIRE (IN newName VARCHAR(10))
BEGIN
  DECLARE newEmp empRow;
  DECLARE maxID INT;

  -- Find the current maximum ID;
  SELECT MAX(id) INTO maxID FROM employee;

  SET (newEmp.id, newEmp.name, newEmp.salary)
    = (maxID + 1, newName, 30000);

  -- Call a procedure to insert the new employee
  CALL ADD_EMP (newEmp);
END@

CREATE PROCEDURE FIRE_EMP (IN empID INT)
BEGIN
  DECLARE emp empRow;

  -- SELECT INTO a row variable
  SELECT * INTO emp FROM employee WHERE id = empID;

  DELETE FROM employee WHERE id = empID;

  INSERT INTO former_employee VALUES (emp.id, emp.name);
END@

CALL NEW_HIRE('Adam')@

CALL FIRE_EMP(1)@

SELECT * FROM employee@

SELECT * FROM former_employee@
```

The following is the output of running the script from the CLP in verbose mode:

```
CREATE TABLE employee (id INT, name VARCHAR(10), salary DECIMAL(9,2))
DB20000I  The SQL command completed successfully.
```

```
INSERT INTO employee VALUES (1, 'Mike', 35000), (2, 'Susan', 35000)
DB20000I  The SQL command completed successfully.

CREATE TABLE former_employee (id INT, name VARCHAR(10))
DB20000I  The SQL command completed successfully.

CREATE TYPE empRow AS ROW ANCHOR ROW OF employee
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE ADD_EMP (IN newEmp empRow)
BEGIN
  INSERT INTO employee VALUES newEmp;
END
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE NEW_HIRE (IN newName VARCHAR(10))
BEGIN
  DECLARE newEmp empRow;
  DECLARE maxID INT;

  -- Find the current maximum ID;
  SELECT MAX(id) INTO maxID FROM employee;

  SET (newEmp.id, newEmp.name, newEmp.salary) = (maxID + 1, newName, 30000);

  -- Call a procedure to insert the new employee
  CALL ADD_EMP (newEmp);
END
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE FIRE_EMPLOYEE (IN empID INT)
BEGIN
  DECLARE emp empRow;

  -- SELECT INTO a row variable
  SELECT * INTO emp FROM employee WHERE id = empID;

  DELETE FROM employee WHERE id = empID;

  INSERT INTO former_employee VALUES (emp.id, emp.name);
END
DB20000I  The SQL command completed successfully.

CALL NEW_HIRE('Adam')

  Return Status = 0

CALL FIRE_EMPLOYEE(1)

  Return Status = 0

SELECT * FROM employee

ID          NAME       SALARY
----------- ---------- -----------
          2 Susan         35000.00
          3 Adam          30000.00

  2 record(s) selected.


SELECT * FROM former_employee

ID          NAME
```

```
----------- ----------
          1 Mike

  1 record(s) selected.
```

**Example: Row data type use in an SQL function:**

Row data types can be used in SQL functions to construct, store, or modify record data.

Variables based on row data types can be used as a simple way to hold a row value that has the same format as a table. When used in this way, it is helpful to initialize the row variable upon its first use.

The following is an example of a DB2 CLP script that contains SQL statements that create a table, a row data type, and a function that includes the declaration of a row variable, a row reference and an invocation of the UDF:

```
CREATE TABLE t1 (deptNo VARCHAR(3),
                 reportNo VARCHAR(3),
                 deptName VARCHAR(29),
                 mgrNo VARCHAR (8),
                 location VARCHAR(128))@

 INSERT INTO t1 VALUES ('123', 'MM1', 'Sales-1', '0112345', 'Miami')@
 INSERT INTO t1 VALUES ('456', 'MM2', 'Sales-2', '0221345', 'Chicago')@
 INSERT INTO t1 VALUES ('789', 'MM3', 'Marketing-1', '0331299', 'Toronto')@

 CREATE TYPE deptRow AS ROW (r_deptNo VARCHAR(3),
                             r_reportNo VARCHAR(3),
                             r_depTName VARCHAR(29),
                             r_mgrNo VARCHAR (8),
                             r_location VARCHAR(128))@

 CREATE FUNCTION getLocation(theDeptNo VARCHAR(3),
                              reportNo VARCHAR(3),
                              theName VARCHAR(29))
  RETURNS VARCHAR(128)
  BEGIN

    -- Declare a row variable
    DECLARE dept deptRow;

    -- Assign values to the fields of the row variable
    SET dept.r_deptno = theDeptNo;
    SET dept.r_reportNo = reportNo;
    SET dept.r_deptname = theName;
    SET dept.r_mgrno = '';
    SET dept.r_location = '';

    RETURN
      (SELECT location FROM t1 WHERE deptNo = dept.r_deptno);

  END@

VALUES (getLocation ('789', 'MM3','Marketing-1'))@
```

When executed this CLP script creates a table, inserts rows into the table, creates a row data type, and a UDF.

The function getLocation is an SQL UDF that declares a row variable and assigns values to it fields using the input parameter values. It references one of the fields in the row variable within the SELECT statement that defines the scalar value returned by the function.

When the VALUES statement is executed at the end of the script, the UDF is invoked and the scalar return value is returned.

The following is the output of running this script from the CLP:

```
CREATE TABLE t1 (deptNo VARCHAR(3), reportNo VARCHAR(3),
 deptName VARCHAR(29), mgrNo VARCHAR (8), location VARCHAR(128))
DB20000I  The SQL command completed successfully.

INSERT INTO t1 VALUES ('123', 'MM1', 'Sales-1', '0112345', 'Miami')
DB20000I  The SQL command completed successfully.

INSERT INTO t1 VALUES ('456', 'MM2', 'Sales-2', '0221345', 'Chicago')
DB20000I  The SQL command completed successfully.

INSERT INTO t1 VALUES ('789', 'MM3', 'Marketing-1', '0331299', 'Toronto')
DB20000I  The SQL command completed successfully.

CREATE TYPE deptRow AS ROW (r_deptNo VARCHAR(3), r_reportNo VARCHAR(3), r_depTNa
me VARCHAR(29), r_mgrNo VARCHAR (8), r_location VARCHAR(128))
DB20000I  The SQL command completed successfully.

CREATE FUNCTION getLocation(theDeptNo VARCHAR(3),
                            reportNo VARCHAR(3),
                            theName VARCHAR(29))
 RETURNS VARCHAR(128)
 BEGIN
   DECLARE dept deptRow;
   SET dept.r_deptno = theDeptNo;
   SET dept.r_reportNo = reportNo;
   SET dept.r_deptname = theName;
   SET dept.r_mgrno = '';
   SET dept.r_location = '';
 RETURN
   (SELECT location FROM t1 WHERE deptNo = dept.r_deptno);
 END
DB20000I  The SQL command completed successfully.

VALUES (getLocation ('789', 'MM3','Marketing-1'))

1

--------------------------------------------------------------------------------
------------------------------------------------
Toronto


  1 record(s) selected.
```

## Array types

An array type is a user-defined data type consisting of an ordered set of elements of a single data type.

An *ordinary* array type has a defined upper bound on the number of elements and uses the ordinal position as the array index.

An *associative* array type has no specific upper bound on the number of elements and each element has an associated index value. The data type of the index value can be an integer or a character string but is the same data type for the entire array.

An array type can be used as the type for global variables, SQL variables, and SQL parameters in SQL PL to provide flexible manipulation of a collection of values of a single data type.

## Comparison of arrays and associative arrays

Simple arrays and associative arrays differ in multiple ways. Understanding the differences can help you to choose the right data type to use.

The following table highlights the differences between arrays and associative arrays:

*Table 1. Comparison of arrays and associative arrays*

| Arrays | Associative arrays |
|---|---|
| The maximum cardinality of a simple array is defined when the simple array is defined. When a value is assigned to index N, the elements with indices between the current cardinality of the array and N are implicitly initialized to NULL. | There is no user-specified maximum cardinality and no elements are initialized when an associative array variable is declared. The maximum cardinality is limited by the available free memory. |
| The index data type for a simple array must be an integer value. | The index type for an associative array can be one of a set of supported data types. |
| The index values in a simple array must be a contiguous set of integer values. | In an associative array the index values can be sparse. |
| The CREATE TYPE statement for a simple array does not require the specification of the array cardinality. For example, in this statement, no cardinality is specified:<br>`CREATE TYPE simple AS INTEGER ARRAY[];` | In the CREATE TYPE statement for an associative array, instead of requiring a specification of the array cardinality, the index data type is required. For example, in this statement, the cardinality for the index data type is specified as INTEGER:<br>`CREATE TYPE assoc AS INTEGER ARRAY[INTEGER];` |
| A first assignment to a simple array results in the initialization of array elements with index values between 1 and the index value assigned to the array. The following compound SQL (compiled) statement contains the declaration of a simple array variable and the assignment of a value to the variable:<br>`BEGIN`<br>`DECLARE mySimpleA simple;`<br>`SET mySimpleA[100] = 123;`<br>`END`<br>After the execution of the assignment statement, the cardinality of `mySimpleA` is 100; the elements with indices with values 1 to 99 are implicitly initialized to NULL. | A first assignment to an associative array results in the initialization of a single element with a single index value. The following compound SQL (compiled) statement contains the declaration of an associative array variable and the assignment of a value to the variable:<br>`BEGIN`<br>`DECLARE myAssocA assoc;`<br>`SET myAssocA[100] = 123;`<br>`END`<br>After the execution of the assignment statement, the cardinality of the array is 1. |

### Example

## Ordinary array data type

An ordinary array data type is a structure that contains an ordered collection of data elements in which each element can be referenced by its ordinal position in the collection.

If N is the cardinality (number of elements) in an array, the ordinal position associated with each element, called the index, is an integer value greater than or equal to 1 and less than or equal to N. All elements in an array have the same data type.

**Features of the array data type:**

The many features of the array data type make it ideal for use in SQL PL logic.

An array type is a data type that is defined as an array of another data type.

Every array type has a maximum cardinality, which is specified on the CREATE TYPE statement. If A is an array type with maximum cardinality M, the cardinality of a value of type A can be any value between 0 and M, inclusive. Unlike the maximum cardinality of arrays in programming languages such as C, the maximum cardinality of SQL arrays is not related to their physical representation. Instead, the maximum cardinality is used by the system at run time to ensure that subscripts are within bounds. The amount of memory required to represent an array value is usually proportional to its cardinality, and not to the maximum cardinality of its type.

When an array is being referenced, all of the values in the array are stored in main memory. Therefore, arrays that contain a large amount of data will consume large amounts of main memory.

Array element values can be retrieved by specifying the element's corresponding index value.

Array data types are useful when you want to store a set of values of a single data type. This set of values can be used to greatly simplify the passing of values to routines, because a single array value can be passed instead of multiple, possibly numerous, individual parameters.

Array data types differ from associative array data types. Whereas array data types are a simple collection of values, associative arrays are conceptually like an array of arrays. That is associative arrays are ordered arrays that contain zero or more subarray elements, such that the array elements are accessed by a primary index and the subarray elements are accessed by a subindex.

**Restrictions on the array data type:**

It is important to note the restrictions on the array data type before you use it or when troubleshooting problems with their declaration or use.

The following restrictions apply to the array data type:

- Use of the array data type in dynamic compound statements is not supported.
- Use of the ARRAY_AGG function outside of SQL procedures is not supported.
- Use of the UNNEST function outside of SQL procedures is not supported.
- Use of parameters of the array data type in external procedures other than Java™ procedures is not supported.
- The casting of an array to any data type other than a user-defined arrays data type is not supported.
- The containment of elements of any data type other than that specified for the array is not supported.
- The casting of an array with a cardinality larger than that of the target array is not supported.
- The use of arrays as parameters or return types in methods is not supported.
- The use of arrays as parameters or return types in sourced or template functions is not supported.

- The use of arrays as parameters or return types in external scalar or external table functions is not supported.
- The use of arrays as parameters or return types in SQL scalar functions, SQL table functions, or SQL row functions is not supported.
- The assignment or casting of the result value of a TRIM_ARRAY function to any data type other than an array is not supported.
- The assignment or casting of the result value of an ARRAY constructor or an ARRAY_AGG function to any data type other than an array is not supported.
-

**Array variables:**

Array variables are variables based on user-defined array data types. Array variables can be declared, assigned a value, set to another value, or passed as a parameter to and from SQL procedures.

Array variables inherit the properties of the array data types upon which they are based. Array variables are used to hold a set of data of the same data type.

Local array variables can be declared within SQL procedures using the DECLARE statement.

Global array variables can be created using the CREATE VARIABLE statement.

**Creating array variables:**

To create array variables you must first create the array type and then declare the local array variable or create the global array variable.

The following topics show you how to create array data types and array variables:

*Creating an array data type (CREATE TYPE statement):*

Creating an array data type is a task that you would perform as a prerequisite to creating a variable of the array data type.

Before you create an array data type, ensure that you have the privileges required to execute the CREATE TYPE statement.

Array data types can only be created in SQL PL contexts where execution of the CREATE TYPE statement is supported.

**Restrictions**

See: "Restrictions on the array data type" on page 24

1. Define the CREATE TYPE statement
   a. Specify a name for the array data type.
   b. Specify the AS keyword followed by the keyword name for the data type of the array element. For example, INTEGER, VARCHAR.
   c. Specify the ARRAY keyword and the domain of the subindices in the array. For example, if you specify 100, the valid indices will be from 1 to 100. This number is the same as the cardinality of the array - the number of elements in the array.
2. Execute the CREATE TYPE statement from a supported interface.

The CREATE type statement should execute successfully and the array type should be created.

**Example 1:**

```
 CREATE TYPE simpleArray AS INTEGER ARRAY[100];
```

This array data type can contain up to 100 integer values indexed by integer values ranging from 1 to 100.
**Example 2:**

```
 CREATE TYPE id_Phone AS VARCHAR(20) ARRAY[100];
```

This array data type can contain up to 100 phone values stored as VARCHAR(20) data type values indexed by integer values ranging from 1 to 100.

After creating the array data type you can declare an array variable.

*Declaring local variables of type array:*

Declaring array data type variables is a task that you perform after creating array data types if you want to be able to temporarily store or pass array data type values.

Before you create a local variable of type row:

- Read: Array data types
- Read: "Restrictions on the array data type" on page 24
- Read: "Creating an array data type (CREATE TYPE statement)" on page 25
- Ensure that you have the privileges required to execute the DECLARE statement.

Declaring array data types can be done in supported contexts including within: SQL procedures, SQL functions, and triggers.

1. Define the DECLARE statement.
   a. Specify a name for the array data type variable.
   b. Specify the name of the array data type that you used when you created the array data type.

   If the array data type was declared using the following CREATE TYPE statement:

   ```
   CREATE TYPE simpleArray AS INTEGER ARRAY[10];
   ```

   You would declare a variable of this data type as follows:

   ```
   DECLARE myArray simpleArray;
   ```

   If the array data type was declared using the following CREATE TYPE statement:

   ```
   CREATE TYPE id_Phone AS VARCHAR(20) ARRAY[100];
   ```

   You would create a variable of this data type as follows:

   ```
   DECLARE id_Phone_Toronto_List  id_Phone;
   ```

   This array can contain up to 100 phone values stored as VARCHAR(20) data type values indexed by integer values ranging from 1 to 100. The variable name indicates that the phone values are Toronto phone numbers.

2. Include the DECLARE statement within a supported context. This can be within a CREATE PROCEDURE, CREATE FUNCTION, or CREATE TRIGGER statement.
3. Execute the statement which contains the DECLARE statement.

The statement should execute successfully.

If the statement does not execute successfully due to an error with the DECLARE statement:

- Verify the SQL statement syntax of the DECLARE statement and execute the statement again.
- Verify that no other variable with the same name has already been declared within the same context.
- Verify that the array data type was created successfully.

After declaring associative array variables, you might want to assign values to them.

**Assigning values to arrays:**

Values can be assigned to arrays in multiple ways. The following topics show you how to assign values to arrays:

*Assigning array values using the subindex and literal values:*

Values can be assigned to associative arrays using subindices and literal values.

- Read: “Ordinary array data type” on page 23
- Read: “Restrictions on the array data type” on page 24
- Privileges required to execute the SET statement

You would perform this task before performing SQL PL that is conditional on the variable having an assigned value or before passing the variable as a parameter to a routine.

1. Define a SET statement.
   a. Specify the array variable name.
   b. Specify the assignment symbol, '='.
   c. Specify the ARRAY keyword and specify within the required brackets sets of paired values.
2. Execute the SET statement.

The following is an example of how to assign element values to an array named, myArray:

```
SET myArray[1]   =  123;
SET myArray[2]   =  124;
...
SET myArray[100] =  223;
```

If the SET statements execute successfully, the array elements have been defined successfully. To validate that the array was created you can attempt to retrieve values from the array.

If the SET statement failed to execute successfully:

- Verify the SQL statement syntax of the SET statement and execute the statement again.
- Verify that the data type was created successfully.

**Retrieving array values:**

Retrieving array values can be done in multiple ways. The following topics show you how to retrieve values from arrays:

*Retrieving array values using an index:*

Retrieving array element values can be done directly by referencing the array and specifying a sub-index value.

The following are prerequisites to this task:
- Read: "Ordinary array data type" on page 23
- Read: "Restrictions on the array data type" on page 24
- Privileges required to execute the SET statement or any SQL statement that contains the array reference

You would perform this task within SQL PL code in order to access values stored within an array. You might access the array element value as part of an assignment (SET) statement or directly within an expression.

1. Define a SET statement.
   a. Specify a variable name of the same data type as the array element.
   b. Specify the assignment symbol, "=".
   c. Specify the name of the array, square brackets, and within the square brackets an index value.
2. Execute the SET statement.

The following is an example of a SET statement that retrieves an array value:

```
SET mylocalVar = myArray[1];
```

If the SET statement executes successfully, the local variable should contain the array element value.

If the SET statement failed to execute successfully:
- Verify the SQL statement syntax of the SET statement and execute the statement again.
- Verify that the variable is of the same data type as the array element.
- Verify that the array was created successfully and currently exists.

*Retrieving the number of array elements:*

Retrieving the number of array elements in a simple array can most easily be done by using the CARDINALITY function and retrieving the maximum allowed size of an array can be done using the MAX_CARDINALITY function.
- Read: "Ordinary array data type" on page 23
- Read: "Restrictions on the array data type" on page 24
- Privileges required to execute the SET statement

You would perform this task within SQL PL code in order to access a count value of the number of elements in an array. You might access the array element value as part of an assignment (SET) statement or access the value directly within an expression.

1. Define a SET statement.
   a. Declare and specify a variable name of type integer that will hold the cardinality value.
   b. Specify the assignment symbol, ‘='.
   c. Specify the name of the CARDINALITY or MAX_CARDINALTIY function and within the required brackets, the name of the array.
2. Execute the SET statement.

If the SET statement executes successfully, the local variable should contain the count value of the number of elements in the array.

The following is an example of two SET statements that demonstrate these assignments:

```
SET card = CARDINALITY(arrayName);

SET maxcard = MAX_CARDINALITY(arrayName);
```

If the SET statement failed to execute successfully:

- Verify the SQL statement syntax of the SET statement and execute the statement again.
- Verify that the local variable is of the integer data type.
- Verify that the array was created successfully and currently exists.

*Retrieving the first and last array elements (FIRST, LAST functions):*

Retrieving the first and last elements in a simple array can most easily be done by using the FIRST and LAST functions.

- Read: “Ordinary array data type” on page 23
- Read: “Restrictions on the array data type” on page 24
- Privileges required to execute the SET statement

You would perform this task within SQL PL code in order to quickly access the first element in an array.

Define a SET statement:

1. Declare and specify a variable that is of the same type as the array element.
2. Specify the assignment symbol, ‘='.
3. Specify the name of the FIRST or LAST function and within the required brackets, the name of the array.

If the SET statement executes successfully, the local variable should contain the value of the first or last (as appropriate) index value in the array.

For an array of phone numbers defined as:

```
firstPhone index    0              1              2              3
           phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

If the following SQL statement is executed:

```
SET firstPhoneIx = FIRST(phones);
```

The variable firstPhoneIx will have the value 0. This would be true even if the element value in this position was NULL.

The following SET statement accesses the element value in the first position in the array:

```
SET firstPhone = A[FIRST(A)]
```

If the SET statement failed to execute successfully:

- Verify the SQL statement syntax of the SET statement and execute the statement again.
- Verify that the local variable is of the correct data type.
- Verify that the array was created successfully and currently exists.

*Retrieving the next and previous array elements:*

Retrieving the next or previous elements in a simple array can most easily be done by using the PREV and NEXT functions.

- Read: “Ordinary array data type” on page 23
- Read: “Restrictions on the array data type” on page 24
- Privileges required to execute the SET statement

You would perform this task within SQL PL code in order to quickly access the immediately adjacent element value in an array.

1. Define a SET statement:
   a. Declare and specify a variable that is of the same type as the array element.
   b. Specify the assignment symbol, ‘='.
   c. Specify the name of the NEXT or PREV function and within the required brackets, the name of the array.
2. Execute the SET statement.

For an array of phone numbers defined as:

```
  firstPhone index  0              1              2              3
             phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

The following SQL statement sets the variable firstPhone to the value 0..

```
SET firstPhone = FIRST(phones);
```

The following SQL statement sets the variable nextPhone to the value 1.

```
SET nextPhone = NEXT(phones, firstPhone);
```

The following SQL statement sets the variable phoneNumber to the value of the phone number at the next position in the array after nextPhone. This is the array element value at index value position 2.

```
SET phoneNumber = phones[NEXT(phones, nextPhone)];
```

If the SET statement failed to execute successfully:

- Verify the SQL statement syntax of the SET statement and execute the statement again.
- Verify that the local variable is of the correct data type.
- Verify that the array was created successfully and currently exists.

**Trimming the array (TRIM_ARRAY function):**

Trimming an array is a task that you would perform using the TRIM_ARRAY function when you want to remove unnecessary array elements from the end of an array.

- Read: Array data types
- Read: Restrictions on array data types
- Privileges required to execute the SET statement

You would perform this task within SQL PL code in order to quickly remove array elements from the end of an array.

1. Define a SET statement:
   a. Declare and specify an array variable that is of the same type as the array to be modified, or re-use the same array variable.
   b. Specify the assignment symbol, ‘='.
   c. Specify the name of the TRIM_ARRAY function and within the required brackets, the name of the array and the number of elements to be trimmed.
2. Execute the SET statement.

If the SET statement executes successfully, the array phones should contain the updated value.

For an array of phone numbers defined as:

```
   phones  index  0       1       2       3
             phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

After executing the following:

```
SET phones = TRIM_ARRAY ( phones,  2 );
```

The array, phones, will be defined as:

```
phones  index  0       1
         phone  '416-223-2233' '416-933-9333'
```

If the SET statement failed to execute successfully:

- Verify the SQL statement syntax of the SET statement and execute the statement again.
- Verify that the local variable is of the correct data type.
- Verify that the array was created successfully and currently exists.

**Deleting an array element (ARRAY_DELETE):**

Deleting an element permanently from an array can be done using the ARRAY_DELETE function.

- Read: Array data types
- Read: Restrictions on array data types
- Privileges required to execute the SET statement

You would perform this task within SQL PL code in order to delete an element in an array.

1. Define a SET statement:
   a. Declare and specify a variable that is of the same type as the array element.
   b. Specify the assignment symbol, ‘='.

   c. Specify the name of the ARRAY_DELETE function and within the required brackets, the name of the array, and the subindices that define the range of the elements to be deleted.
2. Execute the SET statement.

If the SET statement executes successfully, the array phones should contain the updated value.

For an array of phone numbers defined as:

```
phones  index  0             1             2             3
         phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

After executing the following SQL statement:

```
SET phones = ARRAY_DELETE ( phones,  1, 2 );
```

The array, phones, will be defined as:

```
phones  index  0             3
         phone  '416-223-2233' '416-722-7227'
```

If the SET statement failed to execute successfully:

- Verify the SQL statement syntax of the SET statement and execute the statement again.
- Verify that the local variable is of the correct data type.
- Verify that the array was created successfully and currently exists.

**Determining if an array element exists:**

Determining if an array element exists and has a value is a task that can be done using the ARRAY_EXISTS function.

- Read: “Ordinary array data type” on page 23
- Read: “Restrictions on the array data type” on page 24
- Privileges required to execute the IF statement or any SQL statement in which the ARRAY_EXISTS function is referenced.

You would perform this task within SQL PL code in order to determine if an array element exists within an array.

1. Define an IF statement:
   a. Define a condition that includes the ARRAY_EXISTS function.
   b. Specify the THEN clause and include any logic that you want to have performed if the condition is true and add any ELSE caluse values you want.
   c. Close the IF statement with the END IF clause.
2. Execute the IF statement.

For an array of phone numbers defined as:

```
phones  index  0             1             2             3
         phone  '416-223-2233' '416-933-9333' '416-887-8887' '416-722-7227'
```

After executing the following, the variable x will be set to 1.

```
IF (ARRAY_EXISTS(phones, 2)) THEN
  SET x = 1;
END IF;
```

If the SET statement failed to execute successfully:

- Verify the SQL statement syntax of the SET statement and execute the statement again.
- Verify that the local variable is of the correct data type.
- Verify that the array was created successfully and currently exists.

**Array support in SQL procedures:**

SQL procedures support parameters and variables of array types. Arrays are a convenient way of passing transient collections of data between an application and a stored procedure or between two stored procedures.

Within SQL stored procedures, arrays can be manipulated as arrays in conventional programming languages. Furthermore, arrays are integrated within the relational model in such a way that data represented as an array can be easily converted into a table and data in a table column can be aggregated into an array. The examples below illustrate several operations on arrays. Both examples are command line processor (CLP) scripts that use the percentage character (%) as a statement terminator.

**Example 1**

This example shows two procedures, sub and main. Procedure main creates an array of 6 integers using an array constructor. It then passes the array to procedure sum, which computes the sum of all the elements in the input array and returns the result to main. Procedure sum illustrates the use of array subindexing and of the CARDINALITY function, which returns the number of elements in an array.

```
create type intArray as integer array[100] %

create procedure sum(in numList intArray, out total integer)
begin
declare i, n integer;

set n = CARDINALITY(numList);

set i = 1;
set total = 0;

while (i <= n) do
set total = total + numList[i];
set i = i + 1;
end while;

end %

create procedure main(out total integer)
begin
declare numList intArray;

set numList = ARRAY[1,2,3,4,5,6];

call sum(numList, total);

end %
```

**Example 2**

In this example, we use two array data types (intArray and stringArray), and a persons table with two columns (id and name). Procedure processPersons adds three additional persons to the table, and returns an array with the person names that contain letter 'o', ordered by id. The ids and name of the three persons to be added are represented as two arrays (ids and names). These arrays are used as arguments to the UNNEST function, which turns the arrays into a two-column table, whose elements are then inserted into the persons table. Finally, the last set statement in the procedure uses the ARRAY_AGG aggregate function to compute the value of the output parameter.

```
create type intArray as integer array[100] %
create type stringArray as varchar(10) array[100] %

create table persons (id integer, name varchar(10)) %
insert into persons values(2, 'Tom') %
insert into persons values(4, 'Jill') %
insert into persons values(1, 'Joe') %
insert into persons values(3, 'Mary') %

create procedure processPersons(out witho stringArray)
begin
declare ids intArray;
declare names stringArray;

set ids = ARRAY[5,6,7];
set names = ARRAY['Bob', 'Ann', 'Sue'];

insert into persons(id, name)
(select T.i, T.n from UNNEST(ids, names) as T(i, n));

set witho = (select array_agg(name order by id)
from persons
where name like '%o%');
end %
```

## Associative array data type

An associative array data type is a data type used to represent a generalized array with no predefined cardinality. Associative arrays contain an ordered set of zero or more elements of the same data type, where each element is ordered by and can be referenced by an index value.

The index values of associative arrays are unique, are of the same data type, and do not have to be contiguous.

The following topics provide more information about the associative array data type:

**Features of associative arrays:**

The associative array data type is used to represent associative arrays. It has many features which contribute to its utility.

The associative array data type supports the following associative array properties:

- No predefined cardinality is specified for associative arrays. This enables you to continue adding elements to the array without concern for a maximum size which is useful if you do not know in advance how many elements will constitute a set.
- The array index value can be a non integer data type. VARCHAR and INTEGER are supported index values for the associative array index.
- Index values do not have to be contiguous. In contrast to a conventional array which is indexed by position, an associative array is an array that is indexed by values of another data type and there are not necessarily index elements for all

possible index values between the lowest and highest. This is useful if for example you want to create a set that stores names and phone numbers. Pairs of data can be added to the set in any order and be sorted using which ever data item in the pair is defined as the index.

- The elements in an associative array are sorted in ascending order of index values. The insertion order of elements does not matter.
- Associative array data can be accessed and set using direct references or by using a set of available scalar functions.
- Associative arrays are supported in SQL PL contexts.
- Associative arrays can be used to manage and pass sets of values of the same kind in the form of a collection instead of having to:
  - Reduce the data to scalar values and use one-element-at-a-time processing which can cause network traffic problems.
  - Use cursors passed as parameters.
  - Reduce the data into scalar values and reconstitute them as a set using a VALUES clause.

**Restrictions on associative array data types:**

It is important to note the restrictions on the array data type before you use it or when troubleshooting problems with their declaration or use.

The following restrictions apply to the array data type:

- An associative array can only be declared, created, or referenced in SQL PL contexts. The following is a list of SQL PL contexts in which this data type can be used:
  - Parameter to an SQL function that is defined in a module.
  - Parameter to an SQL function that is not defined in a module, but that has a compound SQL (compiled) statement as function body not defined in a module.
  - Return type from an SQL functions that is defined in a module.
  - Return type from an SQL function that is not defined in a module, but that has a compound SQL (compiled) statement as function body.
  - Parameter to an SQL procedure.
  - Local variable declared in an SQL function that is defined in a module.
  - Local variable declared in an SQL function that is not defined in a module, but that has a compound SQL (compiled) statement as function body.
  - Local variable declared in an SQL procedure.
  - Local variable declared in a trigger with a compound SQL (compiled) statement as trigger body.
  - Expressions in SQL statements within compound compiled (SQL) statements.
  - Expressions in SQL statements in SQL PL contexts.
  - Global variable.

  Any use outside of one of the above SQL PL contexts is not valid.
- Associative arrays cannot be the type of a table column.
- NULL is not permitted as an index value.
- The maximum size of an associative array is limited by system resources.
- Associative arrays can not be input to the TRIM_ARRAY function. Associative array values cannot be stored in table columns.

- The MAX_CARDINALITY function is supported for use with associative arrays, but always returns null because associative arrays do not have a specified maximum size.

**Creating an associative array data type:**

Creating an associative array data type is a task that you would perform as a prerequisite to creating a variable of the associative array data type. Associative array data types are created by executing the CREATE TYPE (array) statement.

Ensure you have the privileges required to execute the CREATE TYPE statement.

Associative array data types can only be used in certain contexts.

1. Define the CREATE TYPE statement:
   a. Specify a name for the associative array data type. A good name is one that clearly specifies the type of data stored in the array. For example: Products might be a good name for an array that contains information about products where the array index is the product identifier. As another example, the name y_coordinate might be a good name for an array where the array index is the x coordinate value in a graph function.
   a. Specify the AS keyword followed by the keyword name for the data type of the array elements (e.g. INTEGER).
   b. Specify the ARRAY keyword. Within the square brackets of the ARRAY clause, specify the data type of the array index. Note: With associative arrays, there is no explicit limit on the number of. elements or on the domain of the array index values.
2. Execute the CREATE TYPE statement from a supported interface.

**Example 1:**
The following is an example of a CREATE TYPE statement that creates an array named assocArray with 20 elements and a array index of type VARCHAR.

```
CREATE TYPE assocArray AS INTEGER ARRAY[VARCHAR(20)];
```

**Example 2:**
The following is an example of a basic associative array definition that uses the names of provinces for indices and where the elements are capital cities:

```
CREATE TYPE capitalsArray AS VARCHAR(12) ARRAY[VARCHAR(16)];
```

If the statement executes successfully the array data type is created in the database and the array data type can be referenced..

After creating the array data type you might want to create an associative array variable.

**Declaring associative array variables:**

Declaring associative array variables is a task that you perform after creating associative array data typeso be able to temporarily store or pass associative array data type values. Local variables are declared using the DECLARE statement. Global variables are created using the CREATE VARIABLE statement.

- Read: Associative array data types
- Read: Restrictions on associative array data types
- Read: Creating the associative array data type

- For global variables, you require the privilege to execute the CREATE VARIABLE statement. For local variables, no privileges required to execute the DECLARE statement

Associative array variables can be declared and used in supported contexts to store sets of row data.

1. Define the DECLARE statement for a local variable or the CREATE TYPE statement for a global variable:
   a. Specify a name for the associative array data type.
   b. Specify the name of the associative array data type that you used when you created the associative array data type.
2. Execute the CREATE TYPE statement from a supported interface.

**Example 1:**
Consider an associative array data type defined as:

```
CREATE TYPE Representative_Location AS VARCHAR(20) ARRAY[VARCHAR(30)];
```

To declare a variable of this data type you would use the DECLARE statement as follows:

```
DECLARE RepsByCity Representative_Location;
```

This array can contain up to the maximum number of associative array element values stored as VARCHAR(20) data type values indexed by unique variable character data type values. The variable name indicates that a set of names of sales representatives is indexed by the name of the city that they represent. In this array, no two sales representative names can be represented by the same city which is the array index value.

**Example 2:**
Consider an associative array data type defined to store as element values, the names of capital cities, where the indices are province names:

```
CREATE TYPE capitalsArray AS VARCHAR(12) ARRAY[VARCHAR(16)];
```

To create a variable of this data type you would use the CREATE VARIABLE statement as follows:

```
CREATE VARIABLE capitals capitalsArray;
```

This array can contain up to the maximum number of associative array element values stored as VARCHAR(20) data type values indexed by unique variable character data type values. The variable name indicates that a set of names of sales representatives is indexed by the name of the city that they represent. In this array, no two sales representative names can be represented by the same city which is the array index value.

If the DECLARE statement or CREATE VARIABLE statement executes successfully, the array data type will have been defined successfully and can be referenced. To validate that the associative array variables was created you can assign values to the array or attempt to reference values in the array.

If the DECLARE statement or CREATE VARIABLE statement failed to execute successfully, verify the SQL statement syntax of the DECLARE statement and execute the statement again. See the DECLARE statement.

**Assigning values to arrays using subindices and literal values:**

Once an associative array variable has been created or declared, values can be assigned to it. One way of assigning values to associative arrays is by direct assignment.

- Read: Associative array data types
- Read: Restrictions on associative array data types
- Ensure that an associative array variable is in the current scope of use.

Assigning values to associative array variable elements can be done by using the assignment statement in which the array is named, the index value is specified and the corresponding element value is assigned.

1. Define the assignment statement for an associative array variable.
   - Specify the variable name, the index value, and the element value.
   - Specify another associative array variable.
2. Execute the assignment statement from a supported interface.

**Example 1:**
The following is an example of a variable declaration and a series of assignment statements that define values in the array:

```
DECLARE capitals capitalsArray;

SET capitals['British Columbia'] = 'Victoria';
SET capitals['Alberta'] = 'Edmonton';
SET capitals['Manitoba'] = 'Winnipeg';
SET capitals['Ontario'] = 'Toronto';
SET capitals['Nova Scotia'] = 'Halifax';
```

In the capitals array, the array index values are province names and the associated array element values are the names of the corresponding capital cities. Associative arrays are sorted in ascending order of index value. The order in which values are assigned to associative array elements does not matter.

**Example 2:**
An associative array variable can also be assigned an associative array variable value of the same associative array data type. This can be done using the assignment statement. For example, consider two associative array variables, capitalsA and capitalsB defined as:

```
DECLARE capitalsA capitalsArray;
DECLARE capitalsB capitalsArray;

SET capitalsA['British Columbia'] = 'Victoria';
SET capitalsA['Alberta'] = 'Edmonton';
SET capitalsA['Manitoba'] = 'Winnipeg';
SET capitalsA['Ontario'] = 'Toronto';
SET capitalsA['Nova Scotia'] = 'Halifax';
```

The variable capitalsB can be assigned the value of the variable capitalsA by executing the following assignment statement:

```
SET capitalsB = capitalsA;
```

Once executed, capitalsB will have the same value as capitalsA.

If the assignment statement executes successfully, the value has been successfully assigned and the new variable value can be referenced.

If the statement failed to execute successfully, verify and correct the SQL statement syntax and verify that the variables named are defined before executing the statement again.

# Cursor types

A cursor type can be the built-in data type CURSOR or a user-defined type that is based on the built-in CURSOR data type. A user-defined cursor type can also be defined with a specific row type to restrict the attributes of the result row of the associated cursor.

When a cursor type is associated with a row data structure (specified by a row), it is called a strongly typed cursor. Only result sets that match the definition can be assigned to and stored in a variable of a strongly typed cursor data type. When no result set definition is associated with a cursor data type definition, the cursor data type is said to be weakly typed. Any result set can be stored in a variable of a weakly typed cursor data type.

The cursor data type is only supported for use with SQL PL. It is primarily used to create cursor type definitions that can be used for cursor variable declarations.

This data type can be used to:

- Define cursor variable declarations.
- Simplify the coding of logic within SQL Procedural Language applications. For example, database applications process sets of records called result sets and in some cases the same result set might need to be referenced and processed in different contexts. Passing defined result sets between interfaces can require complex logic. A cursor data type permits the creation of cursor variables which can be used to store result sets, process result sets, and pass result sets as parameters.
- Facilitate the porting to DB2 SQL PL of code which has a similar data type.

Cursor data types must be created using the CREATE TYPE statement. Once this is done variables of this data type can be declared and referenced. Cursor variables can be assigned a row data structure definition, opened, closed, assigned a set of rows from another cursor variable, or be passed as a parameter from SQL procedures.

## Overview of cursor data types

This overview of cursor data types introduces the types of cursor data types, the scope in which they can be used, as well as provides information about the restrictions and privileges that pertain to their use.

**Types of cursor data types:**

There are two main types of cursor data types: weakly-typed cursor data types and strongly-typed cursor data types. The property of being strongly or weakly typed is defined when the data type is created. This property is maintained in variables created of each type.

The characteristics of strongly-typed cursor data types and weakly typed cursor data types are provided here:

**Strongly-typed cursor data types**

A strongly-typed cursor data type is one that is created with a result set definition specified by a row data structure. These data types are called strongly typed, because when result set values are assigned to them the data types of the result sets can be checked. Cursor data type result set definitions can be defined by providing a row type definition. Only result sets that match the definition can be assigned to and stored in a strongly typed cursor data type. Strong type checking is performed at assignment time and if there are any data type mismatches, an error is raised.

Result set definitions for strongly-typed cursor data types can be provided by a row data type definition or an SQL statement definition.

The following is an example of a cursor data type definition that is defined to return a result set with the same row format as the rowType data type:

```
 CREATE TYPE cursorType AS rowType CURSOR@
```

Only result sets that contain columns of data with the same data type definition as the rowType row definition can be successfully assigned to variables declared to be of the cursorType cursor data type.

The following is an example of a cursor data type definition that is defined to return a result set with the same row format as that which defines table T1:

```
 CREATE TABLE T1 (C1 INT)

  CREATE TYPE cursorType AS ANCHOR ROW OF t1 CURSOR;
```

Only result sets that contain columns of data with the same data type definition as the column definition for the table t1 can be successfully assigned to variables declared to be of the cursorType cursor data type.

The row definition associated with a strongly typed cursor can be referenced as the definition of an anchored data type. The following example illustrates this:

```
CREATE TYPE r1 AS ROW (C1 INT);
CREATE TYPE c1 AS RTEST CURSOR;

DECLARE c1 CTEST;
DECLARE r1 ANCHOR ROW OF CV1;
```

A row data type named r1 is defined, a cursor type named c1 associated with the row definition of r1 is defined. The subsequent SQL statements are examples of variable declarations might appear in an SQL procedure. The second variable declaration is for a variable named r1 which is defined to be of the anchored data type - it is anchored to the row type that was used to define the cursor cv1.

**Weakly-typed cursor data types**

A weakly typed cursor type is not associated with any row data type definition. No type checking is performed when values are assigned to weakly typed cursor variables.

There is a system-defined weakly typed cursor data type named CURSOR that can be used to declare weakly typed cursor variables or parameters. The following is an example of a weakly typed cursor variable declaration based on the system-defined weakly typed cursor data type CURSOR:

```
DECLARE cv1 CURSOR;
```

Weakly typed cursor variables are useful when you must store a result set with an unknown row definition.
To return a weakly typed cursor variable as an output parameter, the cursor must be opened.

In this version, variables based on weakly typed cursor data types cannot be referenced as anchored data types.
User-defined weakly typed cursor data types can be defined.

All other characteristics of cursor variables are common for each type of cursor variable.

**Restrictions on cursor data types:**

Restrictions on cursor data types and cursor variables limit cursor variable functionality as well as where cursor variables can be defined and referenced.

The restrictions on cursor data types and variables are important to note before implementing them. The restrictions can be important in determining whether a cursor variable is appropriate for your needs and can be useful to review when troubleshooting errors related to cursor data type and variable use.

The following restrictions apply to cursor data types in this version:

- Cursor data types can only be created as local types in SQL procedures.

The following restrictions apply to cursor variables in this version:

- Cursor variables are not supported for use in applications. Cursor variables can only be declared and referenced in SQL PL contexts.
- Cursor variables are read-only cursors.
- Rows accessed through the use of a cursor variable are not updatable.
- Cursor variables are not scrollable cursors.
- Strongly typed cursor variable columns cannot be referenced as anchored data types.
- There is no support for global cursor variables.
- XML columns cannot be referenced in cursor variable definitions.
- XQuery language statements cannot be used to define strongly-typed cursor result sets.

**Privileges related to cursor data type use:**

Specific privileges related to cursor data types and variables exist to restrict and control who can create them.

To create cursor data types, you require the following privilege:

- Privilege to execute the CREATE TYPE statement to create a cursor data type.

To declare cursor variables based on existing cursor data types, no privileges are required.

To initialize cursor variables, to open the cursor referenced by a cursor variable, or to fetch values from an opened cursor variable reference, you require the same privileges as are required to execute the DECLARE CURSOR statement.

## Cursor variables

Cursor variables are cursors based on predefined cursor data type. Cursor variables can be un-initialized, initialized, assigned a value, set to another value, or passed as a parameter from SQL procedures. Cursor variables inherit the properties of the cursor data types upon which they are based. Cursor variables can be strongly-typed or weakly-typed. Cursor variables hold a reference to the context of the cursor defined by the cursor data type.

Cursor variables can be declared within SQL procedures using the DECLARE statement.

## Cursor predicates

Cursor predicates are SQL keywords that can be used to determine the state of a cursor defined within the current scope. They provide a means for easily referencing whether a cursor is open, closed or if there are rows associated with the cursor.

Cursor predicates can be referenced in SQL and SQL PL statements wherever the status of a cursor can be used as a predicate condition. The cursor predicates that can be used include:

**IS OPEN**

This predicate can be used to determine if the cursor is in an open state. This can be a useful predicate in cases where cursors are passed as parameters to functions and procedures. Before attempting to open the cursor, this predicate can be used to determine if the cursor is already open.

**IS NOT OPEN**

This predicate can be used to determine if the cursor is closed. Its value is the logical inverse of IS OPEN. This predicate can be useful to determine whether a cursor is closed before attempting to actually close the cursor.

**IS FOUND**

This predicate can be used to determine if the cursor contains rows after the execution of a FETCH statement. If the last FETCH statement executed was successful, the IS FOUND predicate value is true. If the last FETCH statement executed resulted in a condition where rows were not found, the result is false. The result is unknown when:

- the value of cursor-variable-name is null
- the underlying cursor of cursor-variable-name is not open
- the predicate is evaluated before the first FETCH action was performed on the underlying cursor
- the last FETCH action returned an error

The IS FOUND predicate can be useful within a portion of SQL PL logic that loops and performs a fetch with each iteration. The predicate can be used to determine if rows remain to be fetched. It provides an efficient alternative to using a condition handler that checks for the error condition that is raised when no more rows remain to be fetched.

An alternative to using IS FOUND is to use IS NOT FOUND which has the opposite value.

### Example

The following script defines an SQL procedure that contains references to these predicates as well as the prerequisite objects required to successfully compile and call the procedure:

```
CREATE TABLE T1 (c1 INT, c2 INT, c3 INT)@

insert into t1 values (1,1,1),(2,2,2),(3,3,3) @
```

```
CREATE TYPE myRowType AS ROW(c1 INT, c2 INT, c3 INT)@

CREATE TYPE myCursorType AS myRowType CURSOR@


CREATE PROCEDURE p(OUT count INT)
LANGUAGE SQL
BEGIN
  DECLARE C1 cursor;
  DECLARE lvarInt INT;

  SET count = -1;
  SET c1 = CURSOR FOR SELECT c1 FROM t1;

  IF (c1 IS NOT OPEN) THEN
     OPEN c1;
  ELSE
     set count = -2;
  END IF;

  set count = 0;
  IF (c1 IS OPEN) THEN

    FETCH c1 into lvarInt;

    WHILE (c1 IS FOUND) DO
      SET count = count + 1;
      FETCH c1 INTO lvarInt;
    END WHILE;
  ELSE
     SET count = 0;
  END IF;

END@

CALL p()@
```

## Creating cursor variables

To create cursor variables you must first create a cursor type and then create a cursor variable based on the type. The following topics show you how to do these tasks:

**Creating cursor data types using the CREATE TYPE statement:**

Creating a cursor data type is a prerequisite to creating a cursor variable. Cursor data types are created using the CREATE TYPE (cursor) statement.

To perform this task you require:

- Privileges to execute the CREATE TYPE (cursor) statement.
- If creating a strongly typed cursor data type, you must either prepare a row specification or base it on an existing row from a table, view, or cursor.

The CREATE TYPE (cursor) statement defines a cursor data type that can be used in SQL PL to declare parameters and local variables of the cursor data type. A strongly typed cursor data type is created if the row-type-name clause is specified in the CREATE TYPE (cursor) statement. A weakly defined cursor data type is created when the row-type-name clause is omitted.

As an alternative to creating a weakly defined cursor data type, the system-defined weakly defined cursor data type CURSOR can be used when declaring cursor variables.

```
CREATE TYPE weakCursorType AS CURSOR@
```

If you want to create a strongly-typed cursor data type, a row data type definition must exist that will define the result set that can be associated with the cursor. A row data type definition can be derived from an explicitly defined row data type, a table or view, or strongly typed cursor. The following is an example of a row type definition:

```
CREATE TYPE empRow AS ROW (name varchar(128), ID varchar(8))@
```

The following is an example of a table definition from which a row type definition can be derived:

```
CREATE TABLE empTable AS ROW (name varchar(128), ID varchar(8))@
```

To define a strongly-typed cursor data type within a database you must successfully execute the CREATE TYPE (CURSOR) statement from any DB2 interface that supports the execution of SQL statements.

1. Formulate a CREATE TYPE (CURSOR) statement:
   a. Specify a name for the type.
   b. Specify a row definition by doing one of: referencing the name of a row data type, specifying that the type should be anchored to a table or view, or anchored to the result set definition associated with an existing strong cursor type.
2. Execute the CREATE TYPE statement from a supported DB2 interface.

If the CREATE TYPE statement executes successfully, the cursor data type is created in the database.

The following is an example of how to create a weakly typed cursor data type that can be associated with result sets with the same format as the empRow row data type:

```
CREATE TYPE cursorType AS empRow CURSOR@
```

The following is an example of how to create a cursor data type that can be associated with result sets with the same format as the table empTable :

```
CREATE TYPE cursorType AS ANCHOR ROW OF empTable@
```

Once the cursor data type is created, cursor variables can be declared based on this data type.

**Declaring local variables of type cursor:**

Local variables of type cursor can be declared once a cursor data type has been created.

A cursor data type definition must exist in the database. Cursor data types are created by successfully executing the CREATE TYPE (CURSOR) statement. The following is an example of a strongly-typed cursor type definition:

```
CREATE TYPE cursorType AS empRow CURSOR;
```

In this version, cursor variables can only be declared as local variables within SQL procedures. Both strongly-typed and weakly-typed cursor variables can be declared.

1. Formulate a DECLARE statement:
   a. Specify a name for the variable.

b. Specify the cursor data type that will define the variable. If the cursor variable is to be weakly-typed, a user-defined weakly typed cursor data type must be specified or the system-defined weakly-typed cursor data type CURSOR. If the cursor variable is to be based on a strongly-typed cursor data type, you can initialize the variable immediately.

The following is an example of how to formulate a DECLARE statement that will define a cursor variable of type cursorType that is not initialized:

```
DECLARE Cv1 cursorType@
```

The following is an example of how to formulate a DECLARE statement that will define a cursor variable Cv2 with a type that is anchored to the type of the existing cursor variable named Cv1:

```
DECLARE Cv2 ANCHOR DATA TYPE TO Cv1@
```

The following is an example of how to formulate a DECLARE statement that will define a weakly-typed cursor variable:

```
DECLARE Cv1 CURSOR@
```

2. Execute the DECLARE statement within a supported context.

If execution of the DECLARE statement is successful, the cursor variable is created.

Once this cursor variable is created, the cursor variable can be assigned values, referenced, or passed as a parameter.

## Assigning values to cursor variables

Result sets can be assigned to cursor variables at different times and in multiple ways using the SET statement.

**Assigning a query result set to a cursor variable**

A result set of a select query can be assigned to a cursor variable by using the SET statement and the CURSOR FOR keywords. The following is an example of how the result set associated with a query on a table named T is assigned to a cursor variable named c1 that has an identical row definition as the table:

If T is defined as:

```
CREATE TABLE T  (C1 INT, C2 INT, C3 INT);
```

If C1 is a strongly-typed cursor variable that was defined as:

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);
CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow;
DECLARE c1 simpleCur;
```

The assignment can be done as follows:

```
SET c1 = CURSOR FOR SELECT * FROM T;
```

The strong type checking will be successful since c1 has a compatible definition to table T. If c1 was a weakly-typed cursor this assignment would also be successful, because no data type checking would be performed.

**Assigning literal values to a cursor variable**

A result set of a select query can be assigned to a cursor variable by using the SET statement and the CURSOR FOR keywords. The following is an example of how the result set associated with a query on a table named T is assigned to a cursor variable named c1 that has an identical row definition as the table.

Let T be a table defined as:

```
CREATE TABLE T  (C1 INT, C2 INT, C3 INT);
```

Let simpleRow be a row type and simpleCur be a cursor type that are respectively created as:

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);
CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow;
```

Let c1 be a strongly-typed cursor variable that is declared within a procedure as:

```
DECLARE c1 simpleCur;
```

The assignment of literal values to cursor c1 can be done as follows:

```
SET c1 = CURSOR FOR VALUES (1, 2, 3);
```

The strong type checking will be successful since the literal values are compatible with the cursor definition. The following is an example of an assignment of literal values that will fail, because the literal data types are incompatible with the cursor type definition:

```
SET c1 = CURSOR FOR VALUES ('a', 'b', 'c');
```

**Assigning cursor variable values to cursor variable values**

A cursor variable value can be assigned to another cursor variable only if the cursor variables have identical result set definitions. For example:

If c1 and c2 are strongly-typed cursor variable that was defined as:

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);

CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow

DECLARE c1 simpleCur;

DECLARE c2 simpleCur;
```

If c2 has been assigned values as follows:

```
SET c2 = CURSOR FOR VALUES (1, 2, 3);
```

The assignment of the result set of c2 to cursor variable c1 can be done as follows:

```
 SET c1 = c2;
```

Once cursor variables have been assigned values, the cursor variables and cursor variables field values can be assigned and referenced.

## Referencing cursor variables

Cursor variables can be referenced in multiple ways as part of cursor operations related to retrieving and accessing a result set or when calling a procedure and passing cursor variables as parameters.

The following statements can be used to reference cursor variables within an SQL PL context:

- CALL
- SET

- OPEN
- FETCH
- CLOSE

The OPEN, FETCH, and CLOSE statements are most often used together when accessing the result set associated with a cursor variable. The OPEN statement is used to initialize the result set associated with the cursor variable. Upon successful execution of this statement, the cursor variable is associated with the result set and the rows in the result set can be accessed. The FETCH statement is used to specifically retrieve the column values in the current row being accessed by the cursor variable. The CLOSE statement is used to end the processing of the cursor variable.

The following is an example of a created row data type definition and an SQL procedure definition that contains a cursor variable definition. Use of the OPEN, FETCH, and CLOSE statements with the cursor variable are demonstrated within the SQL procedure:

```
CREATE TYPE simpleRow AS ROW (c1 INT, c2 INT, c3 INT);

CREATE PROCEDURE P(OUT p1 INT, OUT p2 INT, PUT p3 INT, OUT pRow simpleRow)
LANGUAGE SQL
BEGIN

      CREATE TYPE simpleCur AS CURSOR RETURNS simpleRow
      DECLARE c1 simpleCur;
      DECLARE localVar1 INTEGER;
      DECLARE localVar2 INTEGER;
      DECLARE localVar3 INTEGER;
      DECLARE localRow simpleRow;

      SET c1 = CURSOR FOR SELECT * FROM T;

      OPEN C1;

      FETCH c1 INTO localVar1, localVar2, localVar3;

      FETCH c1 into localRow;

      SET p1 = localVar1;

      SET p2 = localVar2;

      SET p3 = localVar3;

      SET pRow = localRow;

      CLOSE c1;

END;
```

Cursor variables can also be referenced as parameters in the CALL statement. As with other parameters, cursor variable parameters are simply referenced by name. The following is an example of a CALL statement within an SQL procedure that references a cursor variable named curVar which is an output parameter:

```
CALL P2(curVar);
```

## Determining the number of fetched rows for a cursor

Determining the number of rows associated with a cursor can be efficiently done by using the cursor_rowCount scalar function which takes a cursor variable as a parameter and returns an integer value as an output corresponding to the number of rows that have been fetched since the cursor was opened.

The following prerequisites must be met before you use the cursor_rowCount function:

- A cursor data type must be created.
- A cursor variable of the cursor data type must be declared.
- An OPEN statement referencing the cursor must have been executed.

You can use the cursor_rowCount function within SQL PL contexts and would perform this task whenever in your procedural logic it is necessary to access the count of the number of rows that have been fetched for a cursor so far or the total count of rows fetched. The use of the cursor_rowCount function simplifies accessing the fetched row count which otherwise might require that within looping procedural logic you maintain the count with a declared variable and a repeatedly executed SET statement.

**Restrictions**

The cursor_rowCount function can only be used in SQL PL contexts.

1. Formulate an SQL statement with a reference to the cursor_rowCount scalar function. The following is an example of a SET statement that assigns the output of the cursor_rowCount scalar function to a local variable named rows_fetched:
   ```
   SET rows_fetched = CURSOR_ROWCOUNT(curEmp)
   ```
2. Include the SQL statement containing the cursor_rowCount function reference within a supported SQL PL context. This might be, for example, within a CREATE PROCEDURE statement or a CREATE FUNCTION statement and compile the statement.
3.

The statement should compile successfully.

The following is an example of an SQL procedure that includes a reference to the cursor_rowCount function:

```
CREATE PROCEDURE p()
LANGUAGE SQL
BEGIN

SET rows_fetched = CURSOR_ROWCOUNT(curEmp)

END@
```

Execute the SQL procedure or invoke the SQL function.

## Example: Cursor variable use

Referencing examples of cursor variable use an be useful when designing and implementing cursor variables.

**Cursor variable use within an SQL procedure:**

Referencing examples that demonstrate cursor variable use is a good way to learn how and where you can use cursor variables.

This example shows the following:

- CREATE TYPE statement to create a ROW data type
- CREATE TYPE statement to create a strongly-typed cursor based on a row data type specification
- CREATE PROCEDURE statement to create a procedure that has an output cursor parameter
- CREATE PROCEDURE statement to create a procedure that calls another procedure and passes a cursor as an input parameter

A prerequisite to running this example is that the SAMPLE database must exist. To create the sample database, issue the following command from a DB2 Command Window:

```
db2sampl;
```

The following is an example CLP script that demonstrates the core features of cursor variable use within SQL procedures. The script contains a row data type definition, a cursor type definition and two SQL procedure definitions. The procedure P_CALLER contains a cursor variable definition and a call to a procedure named P. The procedure P defines a cursor, opens the cursor and passes the cursor as an output parameter value. The procedure P_CALLER receives the cursor parameter, fetches the cursor value into a local variable, and then sets two output parameter values named edlvel and lastname based on the local variable value.

```
--#SET TERMINATOR @
update command options using c off @
connect to sample @

CREATE TYPE myRowType AS ROW (edlevel SMALLINT, name VARCHAR(128))@

CREATE TYPE myCursorType AS myRowType CURSOR@

CREATE PROCEDURE P(IN pempNo VARCHAR(8), OUT pcv1 CURSOR)
LANGUAGE SQL
BEGIN
  SET pcv1 = CURSOR FOR SELECT edlevel, lastname FROM employee WHERE empNo = pempNo;
  OPEN pcv1;

END@

CREATE PROCEDURE P_CALLER( IN pempNo VARCHAR(8) ,
                           OUT edlevel SMALLINT,
                           OUT lastname VARCHAR(128))
LANGUAGE SQL
BEGIN
  DECLARE rv1 myRowType;
  DECLARE c1 CURSOR;

  CALL P (pempNo,c1);
  FETCH c1 INTO rv1;
  CLOSE c1;

  SET edlevel = rv1.edlevel;
  SET lastname = rv1.name;

END @

CALL P_CALLER('000180',?,?) @
```

When the above script is run, the following output is generated:

```
update command options using c off
DB20000I  The UPDATE COMMAND OPTIONS command completed successfully.
```

```
connect to sample

   Database Connection Information

 Database server        = DB2/LINUXX8664 9.7.0
 SQL authorization ID   = REGRESS5
 Local database alias   = SAMPLE


CREATE TYPE myRowType AS ROW (edlevel SMALLINT, name VARCHAR(128))
DB20000I  The SQL command completed successfully.

CREATE TYPE myCursorType AS myRowType CURSOR@
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE P(IN pempNo VARCHAR(8),OUT pcv1 CURSOR)
LANGUAGE SQL
BEGIN
  SET pcv1 = CURSOR FOR SELECT edlevel, lastname FROM employee WHERE empNo = pempNo;
  OPEN pcv1;

END
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE P_CALLER( IN pempNo VARCHAR(8) ,
                           OUT edlevel SMALLINT,
                           OUT lastname VARCHAR(128))
LANGUAGE SQL
BEGIN
  DECLARE rv1 myRowType;
  DECLARE c1 CURSOR;

  CALL P (pempNo,c1);
  FETCH c1 INTO rv1;
  CLOSE c1;

  SET EDLEVEL = rv1.edlevel;
  SET LASTNAME = rv1.name;

END
DB20000I  The SQL command completed successfully.

CALL P_CALLER('000180',?,?)

  Value of output parameters
  --------------------------
  Parameter Name  : EDLEVEL
  Parameter Value : 17

  Parameter Name  : LASTNAME
  Parameter Value : SCOUTTEN

  Return Status = 0
```

## Boolean data type

The BOOLEAN type is a built-in data type that can only be used for local variables, global variables, parameters, or return types in compound SQL (compiled) statements. A Boolean value represents a truth value of TRUE or FALSE. A Boolean expression or predicate can result in a value of unknown, which is represented as the null value.

### Restrictions on the Boolean data type

It is important to note the restrictions on the Boolean data type before you use it or when troubleshooting problems with their use.

The following restrictions apply to the boolean data type:

- The Boolean data type can only be referenced as:
  - Local variables declared in SQL functions
  - Local variables declared in SQL procedures
  - Local variables declared in triggers with a compound SQL (compiled) statement as trigger body
  - Parameter to SQL functions with a compound SQL (compiled) statement as function body
  - Parameter to SQL procedure with a compound SQL (compiled) statement as procedure body
  - Return type
  - Global variable in a module
- The Boolean data type cannot be used to define the data type of a column in a table or view.
- The system-defined values TRUE and FALSE cannot be referenced as values to be inserted into a table.
- The Boolean data type cannot be referenced in external routines or client applications.
- The Boolean data type cannot be cast to other data types.
- The Boolean data type cannot be returned as a return code value from an SQL procedure.
- Variables of the Boolean data type can only be assigned one of the following values: TRUE, FALSE, or NULL. Numeric or other data type assignments are not supported.
- Selecting or fetching values into variables of the Boolean data type is not supported.
- The Boolean data type cannot be returned in a result set.
- A Boolean variable cannot be used as a predicate. For example, the following SQL clause is not supported:

  ```
  IF (gb) THEN ...
  ```

  Use of predicates is only supported in the SET statement and RETURN statement from a UDF.

If these restrictions prevent you from using this data type consider using an integer data type instead and assign it values such as 1 for TRUE, 0 for FALSE, and -1 for NULL.

## SQL routines

SQL routines are routines that have logic implemented with only SQL statements, including SQL Procedural Language (SQL PL) statements. They are characterized by having their routine-body logic contained within the CREATE statement that is used to create them. This is in contrast with external routines that have their routine logic implemented in a library built form programming source code. In general SQL routines can contain and execute fewer SQL statements than external routines; however they can be every bit as powerful and high performing when implemented according to best practices.

You can create SQL procedures, SQL functions, and SQL methods. Although they are all implemented in SQL, each routine functional type has different features.

## Overview of SQL routines

SQL routines are routines that have logic implemented with only SQL statements, including SQL Procedural Language (SQL PL) statements. They are characterized by having their routine-body logic contained within the CREATE statement that is used to create them. You can create SQL procedures, SQL functions, and SQL methods. Although they are all implemented in SQL, each routine functional type has different features.

Before deciding to implement a SQL routine, it is important that you first understand what SQL routines are, how they are implemented, and used by reading an "Overview of routines". With that knowledge you can then learn more about SQL routine from the following concept topics so that you can make informed decisions about when and how to use them in your database environment:

- SQL procedures
- SQL functions
- Tools for developing SQL routines
- SQL Procedural Language (SQL PL)
- Comparison of SQL PL and inline SQL PL
- SQL PL statements and features
- Supported inline SQL PL statements and features
- Determining when to use SQL procedures or SQL functions
- Restrictions on SQL routines

After having learned about SQL routines, you might want to do one of the following tasks:

- Develop SQL procedures
- Develop SQL functions
- Develop SQL methods

### CREATE statements for SQL routines

SQL routines are created by executing the appropriate CREATE statement for the routine type. In the CREATE statement you also specify the routine body, which for an SQL routine must be composed only of SQL or SQL PL statements. You can use the IBM® DB2 Development Center to help you create, debug, and run SQL procedures. SQL procedures, functions, and methods can also be created using the DB2 command line processor.

SQL procedures, functions, and methods each have a respective CREATE statement. Although the syntax for these statements is different, there are some common elements to them. In each you must specify the routine name, and parameters if there are to be any as well as a return type. You can also specify additional keywords that provide DB2 with information about the logic contained in the routine. DB2 uses the routine prototype and the additional keywords to identify the routine at invocation time, and to execute the routine with the required feature support and best performance possible.

For specific information on creating SQL procedures in the DB2 Development Center or from the Command Line Processor, or on creating functions and methods, refer to the related topics.

## Determining when to use SQL routines or external routines

When implementing routine logic you can choose to implement SQL routines or external routines. There are reasons for choosing each of these two implementations.

To determine when to choose to implement an SQL routine or an external routine, read the following to determine what if any factors might limit your choice.

- Choose to implement SQL routines if:
  - SQL PL and SQL statements provide adequate support to implement the logic that you require.
  - The routine logic consists primarily of SQL statements that query or modify data and performance is a concern. Logic that contains a relatively small amount of control-flow logic relative to the number of SQL statements that query or modify database data will generally perform better with an SQL routine implementation. SQL PL is intended to be used for implementing procedural logic around database operations and not primarily for programming complex logic.
  - The SQL statements that you need to execute can be executed in an external routine implementation.
  - You want to make the modules highly portable between operating system environments and minimize the dependency on programming language code compilers and script interpreters.
  - You want to implement the logic quickly and easily using a high level programming language.
  - You are more comfortable working with SQL than with scripting or programming languages.
  - You want to secure the logic within the database management system.
  - You want to minimize routine maintenance and routine package maintenance upon release upgrades or operating system upgrades.
  - You want to minimize the amount of code required to implement the logic.
  - You want to maximize the safety of the code that is implemented by minimizing the risk of memory management, pointer manipulation, or other common programming pitfalls.
  - You want to benefit from special SQL caching support made available when SQL PL is used.
- Choose to implement an external procedure if:
  - If the routine logic is very complex and consists of few SQL statements and routine performance is a concern. Logic such as a complex math algorithm, that involves a large amount of string manipulation, or that does not access the database will generally perform better with an external routine implementation.
  - If the SQL statements that you need to execute can be executed in an external routine implementation.
  - The routine logic will make operating system calls - this can only be done with external routines.
  - The routine logic must read from or write to files - this can only be done with external routines.
  - Write to the server file system. Do this only with caution.
  - Invoke an application or script that resides on the database server.
  - Issue particular SQL statements that are not supported in SQL procedures.

  - You are more comfortable programming in a programming language other than SQL PL.

By default if SQL routines can meet your needs, use them. Generally it is a requirement to implement complex logic or to access files or scripts on the database server that motivates the decision to use external routines. Particularly since SQL PL is fast and easy to learn and implement.

## Determining when to use SQL procedures or SQL functions

When faced with the choice of implementing logic with SQL PL in an SQL procedure or an SQL function, there are reasons for choosing each of these two implementations.

Read the following to determine when to choose to use an SQL procedure or an SQL function.
Choose to implement an SQL function if:

- Functional requirements can be met by an SQL function and you don't anticipate later requiring the features provided by an SQL procedure.
- Performance is a priority and the logic to be contained in the routine consists only of queries or returns only a single result set.

  When they only contain queries or the return of a single result set an SQL function performs better than a logically equivalent SQL procedure, because of how SQL functions are compiled.

  In SQL procedures, static queries in the form of SELECT statements and full-select statements are compiled individually, such that each query becomes a section of a query access plan in a package when the SQL procedure is created. There is no recompilation of this package until the SQL procedure is recreated or the package is rebound to the database. This means that the performance of the queries is determined based on information available to the database manager at a time earlier than the SQL procedure execution time and hence might not be optimal. Also with an SQL procedure there is also a small overhead entailed when the database manager transfers between executing procedural flow statements and SQL statements that query or modify data.

  SQL functions however are expanded and compiled within the SQL statement that references them which means that they are compiled each time that SQL statement is compiled which depending on the statement might happen dynamically. Because SQL functions are not directly associated with a package, there is no overhead entailed when the database manager transfers between executing procedural flow statements and SQL statements that query or modify data.

Choose to implement an SQL procedure if:

- SQL PL features that are only supported in SQL procedures are required. This includes: output parameter support, use of a cursor, the ability to return multiple result sets to the caller, full condition handling support, transaction and savepoint control, or other features.
- You want to execute non-SQL PL statements that can only be executed in SQL procedures.
- You want to modify data and modifying data is not supported for the type of function you need.

Although it isn't always obvious, you can often easily re-write SQL procedures as SQL functions that perform equivalent logic. This can be an effective way to maximize performance when every little performance improvement counts.

## Determining when to use SQL routines or dynamically prepared compound SQL statements

When determining how to implement an atomic block of SQL PL and other SQL statements you might be faced with a choice between using SQL routines or dynamically prepared compound SQL statements. Although SQL routines internally make use of compound SQL statements, the choice of which to use might depend on other factors.

### Performance

If a dynamically prepared compound SQL statement can functionally meet your needs, using one is preferable, because the SQL statements that appear in dynamically prepared compound SQL statements are compiled and executed as a single block. Also these statements generally perform better than CALL statements to logically equivalent SQL procedures.

At SQL procedure creation time, the procedure is compiled and a package is created. The package contains the best execution path for accessing data as of the SQL procedure compile time. Dynamically prepared compound SQL statements are compiled when they are executed. The best execution path for accessing data for these statements is determined using the most up to date database information which can mean that their access plan can be better than that of a logically equivalent SQL procedure that was created at an earlier time which means that they might perform better.

### Complexity of the required logic

If the logic is quite simple and the number of SQL statements is relatively small, consider using inline SQL PL in a dynamically prepared compound SQL statement (specifying ATOMIC) or in an SQL function. SQL procedures can also handle simple logic, but use of SQL procedures incurs some overhead, such as creating the procedure and calling it, that, if not required, is best avoided.

### Number of SQL statements to be executed

In cases where only one or two SQL statements are to be executed, there might be no benefit in using an SQL procedure. This might actually negatively impact the total performance required to execute these statements. In such a case, it is better to use inline SQL PL in a dynamically prepared compound SQL statement.

### Atomicity and transaction control

Atomicity is another consideration. A compound SQL (inlined) statement must be atomic. Commits and rollbacks are not supported in compound SQL (inlined) statements. If transaction control is required or if support for rollback to a savepoint is required, SQL procedures must be used.

### Security

Security can also be a consideration. SQL procedures can only be executed by users with EXECUTE privilege on the procedure. This can be useful if you need to limit who can execute a particular piece of logic. The ability to execute a dynamically prepared compound SQL statement can also be managed. However SQL procedure execution authorization provides an extra layer of security control.

### Feature support

If you need to return one or more result sets, you must use SQL procedures.

### Modularity, longevity, and re-use

SQL procedures are database objects that are persistently stored in the database and can be consistently referenced by multiple applications or scripts. Dynamically prepared compound SQL statements are not stored in the database and therefore the logic they contain cannot be readily re-used.

If SQL procedures can meet your needs, use them. Generally it is a requirement to implement complex logic or to use the features supported by SQL procedures, but not available to dynamically prepared compound SQL statements that motivates the decision to use SQL procedures.

## Rewriting SQL procedures as SQL user-defined functions

To maximize performance in a database management system, if possible, it can sometimes be beneficial to rewrite simple SQL procedures as SQL functions. Procedures and functions share the fact that their routine-bodies are implemented with a compound block that can contain SQL PL. In both, the same SQL PL statements are included within compound blocks bounded by BEGIN and END keywords.

There are some things to note when translating an SQL procedure into an SQL function:

- The primary and only reason to do this is to improve routine performance when the logic only queries data.
- In a scalar function you might have to declare variables to hold the return value to get around the fact that you cannot directly assign a value to any output parameter of the function. The output value of a user-defined scalar function is only specified in the RETURN statement for the function.
- If an SQL function is going to modify data, it must be explicitly created using the MODIFIES SQL clause so that is can contain SQL statements that modify data.

In the example that follows an SQL procedure and an SQL scalar function that are logically equivalent are shown. These two routines functionally provide the same output value given the same input values, however they are implemented and invoked in slightly different ways.

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR(20),
                           IN Pid INT,
                           OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN

  IF Vendor = 'Vendor 1'
   THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
   THEN SET price = (SELECT Price FROM V2Table
                  WHERE Pid = GetPrice.Pid);
  END IF;
END
```

This procedure takes in two input parameter values and returns an output parameter value that is conditionally determined based on the input parameter values. It uses the IF statement. This SQL procedure is invoked by executing the CALL statement. For example from the CLP, you might execute the following:

```
CALL GetPrice( 'Vendor 1',  9456, ?)
```

The SQL procedure can be rewritten as a logically-equivalent SQL table-function as follows:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
    RETURNS  DECIMAL(10,3)
LANGUAGE SQL  MODIFIES SQL
BEGIN
  DECLARE price DECIMAL(10,3);

  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
  ELSE IF Vendor = 'Vendor 2'
    THEN SET price = (SELECT Price FROM V2Table
             WHERE Pid = GetPrice.Pid);
  END IF;

  RETURN price;
END
```

This function takes in two input parameters and returns a single scalar value, conditionally based on the input parameter values. It requires the declaration and use of a local variable named price to hold the value to be returned until the function returns whereas the SQL procedure can use the output parameter as a variable. Functionally these two routines are performing the same logic.

Now, of course the execution interface for each of these routines is different. Instead of simply calling the SQL procedure with the CALL statement, the SQL function must be invoked within an SQL statement where an expression is allowed. In most cases this isn't a problem and might actually be beneficial if the intention is to immediately operate on the data returned by the routine. Here are two examples of how the SQL function can be invoked.

It can be invoked using the VALUES statement:

```
VALUES (GetPrice('Vendor 1', 9456))
```

It can also be invoked in a SELECT statement that for example might select values from a table and filter rows based on the result of the function:

```
SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 10
```

## SQL procedures

SQL procedures are procedures implemented completely with SQL that can be used to encapsulate logic that can be invoked like a programming sub-routine. There are many useful applications of SQL procedures within a database or database application architecture. SQL procedures can be used to create simple scripts for quickly querying transforming, and updating data or for generating basic reports, for improving application performance, for modularizing applications, and for improving overall database design, and database security.

There are many features of SQL procedures which make them powerful routine options.

Before deciding to implement a SQL procedure, it is important that you understand what SQL procedures are in the context of SQL routines, how they are implemented, and how they can be used, by first learning about routines and then by referring to the topic, "Overview of SQL procedures".

## Features of SQL procedures

SQL procedures are characterized by many features. SQL procedures:

- Can contain SQL Procedural Language statements and features which support the implementation of control-flow logic around traditional static and dynamic SQL statements.
- Are supported in the entire DB2 family brand of database products in which many if not all of the features supported in DB2 Version 9 are supported.
- Are easy to implement, because they use a simple high-level, strongly typed language.
- SQL procedures are more reliable than equivalent external procedures.
- Adhere to the SQL99 ANSI/ISO/IEC SQL standard.
- Support input, output, and input-output parameter passing modes.
- Support a simple, but powerful condition and error-handling model.
- Allow you to return multiple result sets to the caller or to a client application.
- Allow you to easily access the SQLSTATE and SQLCODE values as special variables.
- Reside in the database and are automatically backed up and restored.
- Can be invoked wherever the CALL statement is supported.
- Support nested procedure calls to other SQL procedures or procedures implemented in other languages.
- Support recursion.
- Support savepoints and the rolling back of executed SQL statements to provide extensive transaction control.
- Can be called from triggers.

SQL procedures provide extensive support not limited to what is listed above. When implemented according to best practices, they can play an essential role in database architecture, database application design, and in database system performance.

## Designing SQL procedures

Designing SQL procedures requires an understanding of your requirements, SQL procedure features, how to use the SQL features, and knowledge of any restrictions that might impede your design. The following topics about SQL procedure design will help you learn how to design SQL procedures that make best use of SQL procedure features.

- Parts of SQL procedures
- Cross-platform SQL stored procedure considerations
- Supported SQL PL statements and language features in SQL procedures
- OLTP considerations for SQL procedures
- Performance of SQL procedures
- Rewriting SQL procedures as SQL user-defined functions
- Handling DB2 errors and warnings

**Parts of SQL procedures:** To understand SQL procedures, it helps to understand the parts of an SQL procedure. The following are just some of the parts of SQL procedures:

- Structure of SQL procedures
- Parameters in SQL procedures
- Variables in SQL procedures
- SQLCODE and SQLSTATE in SQL procedures
- Atomic blocks and scope of variables in SQL procedures
- Cursors in SQL procedures
- Logic elements in SQL PL
- Condition and error handlers in SQL procedures
- SQL statements that can be executed in SQL procedures

**Structure of SQL procedures:** SQL procedures consist of several logic parts and SQL procedure development requires you to implement these parts according to a structured format. The format is quite straight-forward and easy to follow and is intended to simplify the design and semantics of routines.

The core of an SQL procedure is a compound statement. Compound statements are bounded by the keywords BEGIN and END. These statements can be ATOMIC or NOT ATOMIC. By default they are NOT ATOMIC.

Within a compound statement, multiple optional SQL PL objects can be declared and referenced with SQL statements. The following diagram illustrates the structured format of a compound statement within SQL procedures:

```
label:  BEGIN
  Variable declarations
  Condition declarations
  Cursor declarations
  Condition handler declarations
  Assignment, flow of control, SQL statements and other compound statements
END label
```

The diagram shows that SQL procedures can consist of one or more optionally atomic compound statements (or blocks) and that these blocks can be nested or serially introduced within a single SQL procedure. Within each of these atomic blocks there is a prescribed order for the optional variable, condition, and handler declarations. These must precede the introduction of procedural logic implemented with SQL-control statements and other SQL statements and cursor declarations. Cursors can be declared anywhere with the set of SQL statements contained in the SQL procedure body.

To clarify control-flow, SQL procedure atomic blocks can be labeled as can many of the SQL control-statements contained within them. This makes it easier to be precise when referencing variables and transfer of control statement references.

Here is an example of an SQL procedure that demonstrates each of the elements listed above:

```
CREATE PROCEDURE DEL_INV_FOR_PROD (IN prod INT, OUT err_buffer VARCHAR(128))
LANGUAGE SQL
DYNAMIC RESULT SETS 1
BEGIN

  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLCODE integer DEFAULT 0;
  DECLARE NO_TABLE CONDITION FOR SQLSTATE '42704';
  DECLARE cur1 CURSOR WITH RETURN TO CALLER
                FOR SELECT * FROM Inv;

  A: BEGIN ATOMIC
```

```
         DECLARE EXIT HANDLER FOR NO_TABLE
           BEGIN
              SET ERR_BUFFER='Table Inv does not exist';
           END;

         SET err_buffer = '';

         IF (prod < 200)
           DELETE FROM Inv WHERE product = prod;
         ELSE IF (prod < 400)
           UPDATE Inv SET quantity = 0 WHERE product = prod;
         ELSE
           UPDATE Inv SET quantity = NULL WHERE product = prod;
         END IF;

     B:  OPEN cur1;

  END
```

### NOT ATOMIC compound statements in SQL procedures

The previous example illustrated a NOT ATOMIC compound statement and is the default type used in SQL procedures. If an unhandled error condition occurs within the compound statement, any work that is completed before the error will not be rolled back, but will not be committed either. The group of statements can only be rolled back if the unit of work is explicitly rolled back using ROLLBACK or ROLLBACK TO SAVEPOINT statements. You can also use the COMMIT statement to commit successful statements if it makes sense to do so.

Here is an example of an SQL procedure with a NOT ATOMIC compound statement:

```
CREATE PROCEDURE not_atomic_proc ()
LANGUAGE SQL
SPECIFIC not_atomic_proc
 nap:  BEGIN NOT ATOMIC

 INSERT INTO c1_sched (class_code, day)
   VALUES ('R11:TAA', 1);

 SIGNAL SQLSTATE '70000';

 INSERT INTO c1_sched (class_code, day)
   VALUES ('R22:TBB', 1);

END nap
```

When the SIGNAL statement is executed it explicitly raises an error that is not handled. The procedure returns immediately afterwards. After the procedure returns, although an error occurred, the first INSERT statement did successfully execute and inserted a row into the c1_sched table. The procedure neither committed, nor rolled back the row insert and this remains to be done for the complete unit of work in which the SQL procedure was called.

### ATOMIC compound statements in SQL procedures

As the name suggests, ATOMIC compound statements, can be thought of as a singular whole. If any unhandled error conditions arise within it, all statements that have executed up to that point are considered to have failed as well and are therefore rolled back.

Atomic compound statements cannot be nested inside other ATOMIC compound statements.

You cannot use the SAVEPOINT statement, the COMMIT statement, or the ROLLBACK statement from within an ATOMIC compound statement. These are only supported in NOT ATOMIC compound statements within SQL procedures.

Here is an example of an SQL procedure with an ATOMIC compound statement:

```
CREATE PROCEDURE atomic_proc ()
LANGUAGE SQL
SPECIFIC atomic_proc

ap:  BEGIN ATOMIC

   INSERT INTO c1_sched (class_code, day)
     VALUES ('R33:TCC', 1);

   SIGNAL SQLSTATE '70000';

   INSERT INTO c1_sched (class_code, day)
     VALUES ('R44:TDD', 1);

END ap
```

When the SIGNAL statement is executed it explicitly raises an error that is not handled. The procedure returns immediately afterwards. The first INSERT statement is rolled back despite successfully executing resulting in a table with no inserted rows for this procedure.

**Labels and SQL procedure compound statements**

Labels can optionally be used to name any executable statement in an SQL procedure, including compound statements and loops. By referencing labels in other statements you can force the flow of execution to jump out of a compound statement or loop or additionally to jump to the beginning of a compound statement or loop. Labels can be referenced by the GOTO, ITERATE, and LEAVE statements.

Optionally you can supply a corresponding label for the END of a compound statement. If an ending label is supplied, it must be same as the label used at its beginning.

Each label must be unique within the body of an SQL procedure.

Labels can also be used to avoid ambiguity if a variable with the same name has been declared in more than one compound statement if the stored procedure. A label can be used to qualify the name of an SQL variable.

**Parameters in SQL procedures:** SQL procedures support parameters for the passing of SQL values into and out of procedures.

Parameters can be useful in SQL procedures when implementing logic that is conditional on a particular input or set of input scalar values or when you need to return one or more output scalar values and you do not want to return a result set.

It is good to understand the features of and limitations of parameters in SQL procedures when designing or creating SQL procedures.

- DB2 supports the optional use of a large number of input, output, and input-output parameters in SQL procedures. The keywords IN, OUT, and INOUT in the routine signature portion of CREATE PROCEDURE statements indicate the mode or intended use of the parameter. IN and OUT parameters are passed by value, and INOUT parameters are passed by reference.
- When multiple parameters are specified for a procedure they must each have a unique name.
- If a variable is to be declared within the procedure with the same name as a parameter, the variable must be declared within a labeled atomic block nested within the procedure. Otherwise DB2 will detect what would otherwise be an ambiguous name reference.
- Parameters to SQL procedures cannot be named either of SQLSTATE or SQLCODE regardless of the data type for the parameter.

Refer to the CREATE PROCEDURE (SQL) statement for complete details about parameter references in SQL procedures.

The following SQL procedure named myparams illustrates the use of IN, INOUT, and OUT parameter modes. Let us say that SQL procedure is defined in a CLP file named myfile.db2 and that we are using the command line.

```
CREATE PROCEDURE myparams (IN p1 INT, INOUT p2 INT, OUT p3 INT)
LANGUAGE SQL
BEGIN
  SET p2 = p1 + 1;
  SET p3 = 2 * p2;
END@
```

**Parameter markers:** A parameter marker, often denoted by a question mark (?) or a colon followed by a variable name (:*var1*), is a place holder in an SQL statement whose value is obtained during statement execution. An application associates parameter markers to application variables. During the execution of the statement, the values of these variables replace each respective parameter marker. Data conversion might take place during the process.

**Benefits of parameter markers**

For SQL statements that need to be executed many times, it is often beneficial to prepare the SQL statement once, and reuse the query plan by using parameter markers to substitute the input values during runtime. In DB2® 9, a parameter marker is represented in one of two ways:

- The first style, with a "?" character, is used in dynamic SQL execution (dynamic Embedded SQL, CLI, Perl, etc).
- The second style represents the embedded SQL standard construction where the name of the variable is prefixed with a colon (:*var1*) . This style is used in static SQL execution and is commonly referred to as a host variable.

Use of either style indicates where an application variable is to be substituted inside an SQL statement. Parameter markers are referenced by number, and are numbered sequentially from left to right, starting at one. Before the SQL statement is executed, the application must bind a variable storage area to each parameter marker specified in the SQL statement. In addition, the bound variables must be a valid storage area, and must contain input data values when the prepared statement is executed against the database.

The following example illustrates an SQL statement containing two parameter markers.

```
SELECT * FROM customers WHERE custid = ? AND lastname = ?
```

### Supported types

DB2 supports untyped parameter markers, which can be used in selected locations of an SQL statement. Table 1 lists the restrictions on parameter marker usage.

*Table 2. Restrictions on parameter marker usage*

| Untyped parameter marker location | Data type |
|---|---|
| Expression: Alone in a select list | Error |
| Expression: Both operands of an arithmetic operator | Error |
| Predicate: Left-hand side operand of an IN predicate | Error |
| Predicate: Both operands of a relational operator | Error |
| Function: Operand of an aggregation function | Error |

### Examples

DB2® provides a rich set of standard interfaces including CLI/ODBC, JDBC, and ADO.NET to access data efficiently. The following code snippets show the use of prepared statement with parameter markers for each data access API.

Consider the following table schema for table t1, where column c1 is the primary key for table t1.

*Table 3. Example table schema*

| Column name | DB2 data type | Nullable |
|---|---|---|
| c1 | INTEGER | false |
| c2 | SMALLINT | true |
| c3 | CHAR(20) | true |
| c4 | VARCHAR(20) | true |
| c5 | DECIMAL(8,2) | true |
| c6 | DATE | true |
| c7 | TIME | true |
| c8 | TIMESTAMP | true |
| c9 | BLOB(30) | true |

The following examples illustrate how to insert a row into table t1 using a prepared statement.

### CLI Example

```
void parameterExample1(void)
{
   SQLHENV henv;
```

```
SQLHDBC hdbc;
SQLHSTMT hstmt;
SQLRETURN rc;
TCHAR server[] = _T("C:\\mysample\\");
TCHAR uid[] = _T("db2e");
TCHAR pwd[] = _T("db2e");
long p1 = 10;
short p2 = 100;
TCHAR p3[100];
TCHAR p4[100];
TCHAR p5[100];
TCHAR p6[100];
TCHAR p7[100];
TCHAR p8[100];
char  p9[100];
long len = 0;

_tcscpy(p3, _T("data1"));
_tcscpy(p4, _T("data2"));
_tcscpy(p5, _T("10.12"));
_tcscpy(p6, _T("2003-06-30"));
_tcscpy(p7, _T("12:12:12"));
_tcscpy(p8, _T("2003-06-30-17.54.27.710000"));

memset(p9, 0, sizeof(p9));
p9[0] = 'X';
p9[1] = 'Y';
p9[2] = 'Z';

rc = SQLAllocEnv(&henv);
// check return code ...

rc = SQLAllocConnect(henv, &hdbc);
// check return code ...

rc = SQLConnect(hdbc, (SQLTCHAR*)server, SQL_NTS,
 (SQLTCHAR*)uid, SQL_NTS, (SQLTCHAR*)pwd, SQL_NTS);
// check return code ...

rc = SQLAllocStmt(hdbc, &hstmt);
// check return code ...

// prepare the statement
rc = SQLPrepare(hstmt, _T("INSERT INTO t1 VALUES (?,?,?,?,?,?,?,?,?)"), SQL_NTS);
// check return code ...

// bind input parameters
rc = SQLBindParameter(hstmt, (unsigned short)1, SQL_PARAM_INPUT,
 SQL_C_LONG, SQL_INTEGER, 4, 0, &p1, sizeof(p1), &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)2, SQL_PARAM_INPUT, SQL_C_LONG,
  SQL_SMALLINT, 2, 0, &p2, sizeof(p2), &len);
// check return code ...

len = SQL_NTS;
rc = SQLBindParameter(hstmt, (unsigned short)3, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_CHAR, 0, 0, &p3[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)4, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_VARCHAR, 0, 0, &p4[0], 100, &len);
// check return code ...

rc = SQLBindParameter(hstmt, (unsigned short)5, SQL_PARAM_INPUT, SQL_C_TCHAR,
  SQL_DECIMAL, 8, 2, &p5[0], 100, &len);
// check return code ...
```

```
   rc = SQLBindParameter(hstmt, (unsigned short)6, SQL_PARAM_INPUT, SQL_C_TCHAR,
     SQL_TYPE_DATE, 0, 0, &p6[0], 100, &len);
   // check return code ...

   rc = SQLBindParameter(hstmt, (unsigned short)7, SQL_PARAM_INPUT, SQL_C_TCHAR,
     SQL_TYPE_TIME, 0, 0, &p7[0], 100, &len);
   // check return code ...

   rc = SQLBindParameter(hstmt, (unsigned short)8, SQL_PARAM_INPUT, SQL_C_TCHAR,
     SQL_TYPE_TIMESTAMP, 0, 0, &p8[0], 100, &len);
   // check return code ...

   len = 3;
   rc = SQLBindParameter(hstmt, (unsigned short)9, SQL_PARAM_INPUT, SQL_C_BINARY,
     SQL_BINARY, 0, 0, &p9[0], 100, &len);
   // check return code ...

   // execute the prepared statement
   rc = SQLExecute(hstmt);
   // check return code ...

   rc = SQLFreeStmt(hstmt, SQL_DROP);
   // check return code ...

   rc = SQLDisconnect(hdbc);
   // check return code ...

   rc = SQLFreeConnect(hdbc);
   // check return code ...

   rc = SQLFreeEnv(henv);
   // check return code ...
```

### C Example

```
EXEC SQL BEGIN DECLARE SECTION;
  char hostVarStmt1[50];
  short hostVarDeptnumb;
EXEC SQL END DECLARE SECTION;

/* prepare the statement with a parameter marker */
strcpy(hostVarStmt1, "DELETE FROM org WHERE deptnumb = ?");
EXEC SQL PREPARE Stmt1 FROM :hostVarStmt1;

/* execute the statement for hostVarDeptnumb = 15 */
hostVarDeptnumb = 15;
EXEC SQL EXECUTE Stmt1 USING :hostVarDeptnumb;
```

### JDBC Example

```
public static void parameterExample1() {

   String driver = "com.ibm.db2e.jdbc.DB2eDriver";
   String url    = "jdbc:db2e:mysample";
   Connection conn = null;
   PreparedStatement pstmt = null;

   try
   {
      Class.forName(driver);

      conn = DriverManager.getConnection(url);

      // prepare the statement
      pstmt = conn.prepareStatement("INSERT INTO t1 VALUES
                                             (?, ?, ?, ?, ?, ?, ?, ?, ?)");
      // bind the input parameters
      pstmt.setInt(1, 1);
      pstmt.setShort(2, (short)2);
      pstmt.setString(3, "data1");
      pstmt.setString(4, "data2");
      pstmt.setBigDecimal(5, new java.math.BigDecimal("12.34"));
      pstmt.setDate(6, new java.sql.Date(System.currentTimeMillis() ) );
      pstmt.setTime(7, new java.sql.Time(System.currentTimeMillis() ) );
      pstmt.setTimestamp (8, new java.sql.Timestamp(System.currentTimeMillis() ) );
      pstmt.setBytes(9, new byte[] { (byte)'X', (byte)'Y', (byte)'Z' } );

      // execute the statement
      pstmt.execute();

      pstmt.close();

      conn.close();
   }
   catch (SQLException sqlEx)
   {
      while(sqlEx != null)
      {
          System.out.println("SQLERROR: \n" + sqlEx.getErrorCode() +
             ", SQLState: " + sqlEx.getSQLState() +
             ", Message: " + sqlEx.getMessage() +
             ", Vendor: " + sqlEx.getErrorCode() );
         sqlEx = sqlEx.getNextException();
      }
   }
   catch (Exception ex)
   {
      ex.printStackTrace();
   }
}
```

## ADO.NET Example [C#]

```
public static void ParameterExample1()
{
   DB2eConnection conn = null;
   DB2eCommand cmd  = null;
   String connString   = @"database=.\; uid=db2e; pwd=db2e";
   int i = 1;

   try
   {
      conn = new DB2eConnection(connString);

      conn.Open();

      cmd = new DB2eCommand("INSERT INTO t1 VALUES
                                      (?, ?, ?, ?, ?, ?, ?, ?, ?)", conn);

      // prepare the command
      cmd.Prepare();

      // bind the input parameters
      DB2eParameter p1 = new DB2eParameter("@p1", DB2eType.Integer);
      p1.Value = ++i;
      cmd.Parameters.Add(p1);

      DB2eParameter p2 = new DB2eParameter("@p2", DB2eType.SmallInt);
      p2.Value = 100;
      cmd.Parameters.Add(p2);

      DB2eParameter p3 = new DB2eParameter("@p3", DB2eType.Char);
      p3.Value = "data1";
      cmd.Parameters.Add(p3);

      DB2eParameter p4 = new DB2eParameter("@p4", DB2eType.VarChar);
      p4.Value = "data2";
      cmd.Parameters.Add(p4);

      DB2eParameter p5 = new DB2eParameter("@p5", DB2eType.Decimal);
      p5.Value = 20.25;
      cmd.Parameters.Add(p5);

      DB2eParameter p6 = new DB2eParameter("@p6", DB2eType.Date);
      p6.Value = DateTime.Now;
      cmd.Parameters.Add(p6);

      DB2eParameter p7 = new DB2eParameter("@p7", DB2eType.Time);
      p7.Value = new TimeSpan(23, 23, 23);
      cmd.Parameters.Add(p7);

      DB2eParameter p8 = new DB2eParameter("@p8", DB2eType.Timestamp);
      p8.Value = DateTime.Now;
      cmd.Parameters.Add(p8);

      byte []barr = new byte[3];
      barr[0] = (byte)'X';
      barr[1] = (byte)'Y';
      barr[2] = (byte)'Z';

      DB2eParameter p9 = new DB2eParameter("@p9", DB2eType.Blob);
      p9.Value = barr;
      cmd.Parameters.Add(p9);

      // execute the prepared command
      cmd.ExecuteNonQuery();
   }
   catch (DB2eException e1)
   {
      for (int i=0; i < e1.Errors.Count; i++)
      {
         Console.WriteLine("Error #" + i + "\n" +
            "Message: " + e1.Errors[i].Message + "\n" +
            "Native: " + e1.Errors[i].NativeError.ToString() + "\n" +
            "SQL: " + e1.Errors[i].SQLState + "\n");
      }
   }
   catch (Exception e2)
   {
       Console.WriteLine(e2.Message);
   }
   finally
   {
      if (conn != null && conn.State != ConnectionState.Closed)
      {
            conn.Close();
            conn = null;
      }
   }
}
```

**Variables in SQL procedures (DECLARE, SET statements):** Local variable support in SQL procedures allows you to assign and retrieve SQL values in support of SQL procedure logic.

Variables in SQL procedures are defined by using the DECLARE statement.

Values can be assigned to variables using the SET statement or the SELECT INTO statement or as a default value when the variable is declared. Literals, expressions, the result of a query, and special register values can be assigned to variables.

Variable values can be assigned to SQL procedure parameters, other variables in the SQL procedure, and can be referenced as parameters within SQL statements that executed within the routine.

The following example demonstrates various methods for assigning and retrieving variable values.

```
CREATE PROCEDURE proc_vars()
SPECIFIC proc_vars
LANGUAGE SQL
BEGIN

  DECLARE v_rcount INTEGER;

  DECLARE v_max DECIMAL (9,2);

  DECLARE v_adate, v_another  DATE;

  DECLARE v_total INTEGER DEFAULT 0;           -- (1)

  DECLARE v_rowsChanged BOOLEAN DEFAULT FALSE; -- (2)

  SET v_total = v_total + 1                    -- (3)

  SELECT MAX(salary)                           -- (4)
    INTO v_max FROM employee;

  VALUES CURRENT_DATE INTO v_date;             -- (5)

  SELECT CURRENT DATE, CURRENT DATE            -- (6)
       INTO v_adate, v_another
  FROM SYSIBM.SYSDUMMY1;

  DELETE FROM T;
  GET DIAGNOSTICS v_rcount = ROW_COUNT;        -- (7)

  IF v_rcount > 0 THEN                         -- (8)
    SET is_done = TRUE;
  END IF;
END
```

When declaring a variable, you can specify a default value using the DEFAULT clause as in line (1). Line (2) shows the declaration of a variable of the Boolean data type with a default value of FALSE. Line (3) shows that a SET statement can be used to assign a single variable value. Variables can also be set by executing a SELECT or FETCH statement in combination with the INTO clause as shown in line (4). Lines (5) and (6) show how the VALUES INTO statement can be used to evaluate a function or special register and assign the value to a variable or to multiple variables.

You can also assign the result of a GET DIAGNOSTICS statement to a variable. GET DIAGNOSTICS can be used to get a handle on the number of affected rows (updated for an UPDATE statement, DELETE for a DELETE statement) or to get the return status of a just executed SQL statement. Line (7) shows how the number of rows modified by the just previously executed DELETE statement can be assigned to a variable.

Line (8) demonstrates how a piece of logic can be used to determine the value to be assigned to a variable. In this case, if rows were changed as part of the earlier DELETE statement and the GET DIAGNOSTICS statement execution resulted in the variable v_rcount being assigned a value greater than zero, the variable is_done is assigned the value TRUE.

**SQLCODE and SQLSTATE variables in SQL procedures:** To perform error handling or to help you debug your SQL procedures, you might find it useful to test the value of the SQLCODE or SQLSTATE values, return these values as output parameters or as part of a diagnostic message string, or insert these values into a table to provide basic tracing support.

To use the SQLCODE and SQLSTATE values within SQL procedures, you must declare the following SQL variables in the SQL procedure body:

```
   DECLARE SQLCODE INTEGER DEFAULT 0;
   DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

DB2 implicitly sets these variables whenever a statement is executed. If a statement raises a condition for which a handler exists, the values of the SQLSTATE and SQLCODE variables are available at the beginning of the handler execution. However, the variables are reset as soon as the first statement in the handler is executed. Therefore, it is common practice to copy the values of SQLSTATE and SQLCODE into local variables in the first statement of the handler. In the following example, a CONTINUE handler for any condition is used to copy the SQLCODE variable into another variable named `retcode`. The variable `retcode` can then be used in the executable statements to control procedural logic, or pass the value back as an output parameter.

```
BEGIN
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE retcode INTEGER DEFAULT 0;

  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION, SQLWARNING, NOT FOUND
     SET retcode = SQLCODE;

  executable-statements
END
```

**Note:** When you access the SQLCODE or SQLSTATE variables in an SQL procedure, DB2 sets the value of SQLCODE to 0 and SQLSTATE to ‘00000' for the subsequent statement.

**Compound statements and scope of variables in SQL procedures:** SQL procedures can contain one or more compound statements. They can be introduced in serial or can be nested within another compound statement. Each compound statement introduces a new scope in which variables might or might not be available for use.

The use of labels to identify a compound statement is important as the label can be used to qualify and uniquely identify variables declared within the compound statement. This is particularly important when referencing of variables in different compound statements or in nested compound statements.

In the following example there are two declarations of the variable *a*. One instance of it is declared in the outer compound statement that is labelled by *lab1*, and the second instance is declared in the inner compound statement labelled by *lab2*. As it is written, DB2 will presume that the reference to *a* in the assignment-statement is the one which is in the local scope of the compound block, labelled by *lab2*.

However, if the intended instance of the variable *a* is the one declared in the compound statement block labeled with *lab1*, then to correctly reference it in the innermost compound block, the variable should be qualified with the label of that block. That is, it should be qualified as: *lab1.a*.

```
CREATE PROCEDURE P1 ()
LANGUAGE SQL
  lab1: BEGIN
    DECLARE a INT DEFAULT 100;
    lab2: BEGIN
      DECLARE a INT DEFAULT NULL;

      SET a = a + lab1.a;

      UPDATE T1
       SET T1.b = 5
        WHERE T1.b = a;  <-- Variable a refers to lab2.a
                             unless qualified otherwise

    END lab2;
 END lab1
```

The outermost compound statement in an SQL procedure can be declared to be atomic, by adding the keyword ATOMIC after the BEGIN keyword. If any error occurs in the execution of the statements that comprise the atomic compound statement, then the entire compound statement is rolled back.

**Cursors in SQL procedures:** In SQL procedures, a cursor make it possible to define a result set (a set of data rows) and perform complex logic on a row by row basis. By using the same mechanics, an SQL procedure can also define a result set and return it directly to the caller of the SQL procedure or to a client application.

A cursor can be viewed as a pointer to one row in a set of rows. The cursor can only reference one row at a time, but can move to other rows of the result set as needed.

To use cursors in SQL procedures, you need to do the following:

1. Declare a cursor that defines a result set.
2. Open the cursor to establish the result set.
3. Fetch the data into local variables as needed from the cursor, one row at a time.
4. Close the cursor when done

To work with cursors you must use the following SQL statements:

- DECLARE CURSOR
- OPEN
- FETCH
- CLOSE

The following example demonstrates the basic use of a read-only cursor within an SQL procedure:

```
CREATE PROCEDURE sum_salaries(OUT sum INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE p_sum INTEGER;
  DECLARE p_sal INTEGER;
  DECLARE c CURSOR FOR SELECT SALARY FROM EMPLOYEE;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';

   SET p_sum = 0;
```

```
    OPEN c;

    FETCH FROM c INTO p_sal;

    WHILE(SQLSTATE = '00000') DO
       SET p_sum = p_sum + p_sal;
       FETCH FROM c INTO p_sal;
    END WHILE;

    CLOSE c;

    SET sum = p_sum;

 END%
```

Here is a more complex example of use of a cursor within an SQL procedure. This example demonstrates the combined use of a cursor and SQL PL statements.

**SQL PL logic elements in the SQL-procedure body:** Sequential execution is the most basic path that program execution can take. With this method, the program starts execution at the first line of the code, followed by the next, and continues until the final statement in the code has been executed. This approach works fine for very simple tasks, but tends to lack usefulness because it can only handle one situation. Programs often need to be able to decide what to do in response to changing circumstances. By controlling a code's execution path, a specific piece of code can then be used to intelligently handle more than one situation.

SQL PL provides support for variables and flow of control statements that can be used to control the sequence of statement execution. Statements such as IF and CASE are used to conditionally execute blocks of SQL PL statements, while other statements, such as WHILE and REPEAT, are typically used to execute a set of statements repetitively until a task is complete.

Although there are many types of SQL PL statements, there are a few categories into which these can be sorted:

- Variable related statements
- Conditional statements
- Loop statements
- Transfer of control statements

*Variable related statements in SQL procedures:* Variable related SQL statements are used to declare variables and to assign values to variables. There are a few types of variable related statements:

- DECLARE <variable> statement in SQL procedures
- DECLARE <condition> statement in SQL procedures
- DECLARE <condition handler> statement in SQL procedures
- DECLARE CURSOR in SQL procedures
- SET (assignment-statement) in SQL procedures

These statements provide the necessary support required to make use of the other types of SQL PL statements and SQL statements that will make use of variable values.

*Conditional statements in SQL procedures:* Conditional statements are used to define what logic is to be executed based on the status of some condition being satisfied. There are two types of conditional statements supported in SQL procedures:

- CASE
- IF

These statements are similar; however the CASE statements extends the IF statement.

*CASE statement in SQL procedures:* CASE statements can be used to conditionally enter into some logic based on the status of a condition being satisfied. There are two types of CASE statements:

- Simple case statement: used to enter into some logic based on a literal value
- Searched case statement: used to enter into some logic based on the value of an expression

The WHEN clause of the CASE statement defines the value that when satisfied determines the flow of control.

Here is an example of an SQL procedure with a CASE statement with a simple-case-statement-when-clause:

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept)
LANGUAGE SQL
BEGIN

  DECLARE v_workdept CHAR(3);
  SET v_workdept = p_workdept;

  CASE  v_workdept
    WHEN 'A00' THEN
      UPDATE department SET deptname = 'D1';
    WHEN 'B01' THEN
      UPDATE department SET deptname = 'D2';
    ELSE
      UPDATE department SET deptname = 'D3';
    END CASE

END
```

Here is an example of CASE statement with a searched-case-statement-when-clause:

```
CREATE PROCEDURE UPDATE_DEPT (IN p_workdept)
LANGUAGE SQL
BEGIN

  DECLARE v_workdept CHAR(3);
  SET v_workdept = p_workdept;

  CASE
    WHEN v_workdept = 'A00' THEN
      UPDATE department SET deptname = 'D1';
    WHEN v_workdept = 'B01' THEN
      UPDATE department SET deptname = 'D2';
    ELSE
      UPDATE department SET deptname = 'D3';
  END CASE

END
```

The examples provided above are logically equivalent, however it is important to note that CASE statements with a searched-case-statement-when-clause can be very powerful. Any supported SQL expression can be used here. These expressions can contain references to variables, parameters, special registers, and more.

*IF statement in SQL procedures:* IF statements can be used to conditionally enter into some logic based on the status of a condition being satisfied. The IF statement is logically equivalent to a CASE statements with a searched-case-statement-when clause.

The IF statement supports the use of optional ELSE IF clauses and a default ELSE clause. An END IF clause is required to indicate the end of the statement.

Here is an example of procedure that contains an IF statement:

```
CREATE PROCEDURE UPDATE_SAL (IN empNum CHAR(6),
                             INOUT rating SMALLINT)
LANGUAGE  SQL
BEGIN
  IF rating = 1 THEN
    UPDATE employee
    SET salary = salary * 1.10, bonus = 1000
      WHERE empno = empNum;
  ELSEIF rating = 2 THEN
    UPDATE employee
    SET salary = salary * 1.05, bonus = 500
      WHERE empno = empNum;
  ELSE
    UPDATE employee
    SET salary = salary * 1.03, bonus = 0
      WHERE empno = empNum;
  END IF;
END
```

*Looping statements in SQL procedures:* Looping statements provide support for repeatedly executing some logic until a condition is met. The following looping statements are supported in SQL PL:

- FOR
- LOOP
- REPEAT
- WHILE

The FOR statement is distinct from the others, because it is used to iterate over rows of a defined result set, whereas the others are using for iterating over a series of SQL statements until for each a condition is satisfied.

Labels can be defined for all loop-control-statements to identify them.

*FOR statement in SQL procedures:* FOR statements are a special type of looping statement, because they are used to iterate over rows in a defined read-only result set. When a FOR statement is executed a cursor is implicitly declared such that for each iteration of the FOR-loop the next row is the result set if fetched. Looping continues until there are no rows left in the result set.

The FOR statement simplifies the implementation of a cursor and makes it easy to retrieve a set of column values for a set of rows upon which logical operations can be performed.

Here is an example of an SQL procedure that contains only a simple FOR statement:

```
CREATE PROCEDURE P()
LANGUAGE SQL
BEGIN ATOMIC
  DECLARE fullname CHAR(40);

  FOR v AS cur1 CURSOR FOR
            SELECT firstnme, midinit, lastname FROM employee
  DO
    SET fullname = v.lastname || ',' || v.firstnme
                   ||' ' || v.midinit;
    INSERT INTO tnames VALUES (fullname);
  END FOR;
END
```

Note: Logic such as is shown in the example above would be better implemented using the CONCAT function. The simple example serves to demonstrate the syntax.

The for-loop-name specifies a label for the implicit compound statement generated to implemented the FOR statement. It follows the rules for the label of a compound statement. The for-loop-name can be used to qualify the column names in the result set as returned by the select-statement.

The cursor-name simply names the cursor that is used to select the rows from the result set. If it is not specified, the DB2 database manager will automatically generate a unique cursor name internally.

The column names of the select statement must be unique and a FROM clause specifying a table (or multiple tables if doing some kind of JOIN or UNION) is required. The tables and columns referenced must exist prior to the loop being executed. Global temporary tables and declared temporary tables can be referenced.

Positioned updates and deletes, and searched updates and deletes are supported in the FOR loop. To ensure correct results, the FOR loop cursor specification must include a FOR UPDATE clause.

The cursor that is created in support of the FOR statement cannot be referenced outside of the FOR loop.

*LOOP statement in SQL procedures:* The LOOP statement is a special type of looping statement, because has no terminating condition clause. It defines a series of statements that are executed repeatedly until another piece of logic, generally a transfer of control statement, forces the flow of control to jump to some point outside of the loop.

The LOOP statement is generally used in conjunction with one of the following statements: LEAVE, GOTO, ITERATE, or RETURN. These statements can force control to just after the loop, to a specified location in the SQL procedure, to the start of the loop to begin another iteration of the loop, or to exit the SQL procedure. To indicate where to pass flow to when using these statements, labels are used.

The LOOP statement is useful when you have complicated logic in a loop which you might need to exit in more than one way, however it should be used with care to avoid instances of infinite loops.

If the LOOP statement is used alone without a transfer of control statement, the series of statements included in the loop will be executed indefinitely or until a database condition occurs that raises a condition handler that forces a change in the control flow or a condition occurs that is not handled that forces the return of the SQL procedure.

Here is an example of an SQL procedure that contains a LOOP statement. It also uses the ITERATE and LEAVE statements.

```
CREATE  PROCEDURE ITERATOR()
LANGUAGE SQL
BEGIN
  DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
  DECLARE at_end INTEGER DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';

  DECLARE c1 CURSOR FOR SELECT deptno, deptname
                        FROM department ORDER BY deptno;
  DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;
  OPEN c1;

  ins_loop: LOOP

    FETCH c1 INTO v_deptno, v_deptname;

    IF at_end = 1 THEN
      LEAVE ins_loop;
    ELSEIF v_dept = 'D11' THEN
      ITERATE ins_loop;
    END  IF;

    INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

  END LOOP;

  CLOSE c1;
END
```

*WHILE statement in SQL procedures:* The WHILE statement defines a set of statements to be executed until a condition that is evaluated at the beginning of the WHILE loop is false. The while-loop-condition (an expression) is evaluated before each iteration of the loop.

Here is an example of an SQL procedure with a simple WHILE loop:

```
CREATE PROCEDURE sum_mn (IN p_start INT
                        ,IN p_end INT
                        ,OUT p_sum INT)
SPECIFIC sum_mn
LANGUAGE SQL
smn: BEGIN

DECLARE v_temp INTEGER DEFAULT 0;
DECLARE v_current INTEGER;

SET v_current = p_start;

WHILE (v_current <= p_end) DO
  SET v_temp = v_temp + v_current;
  SET v_current = v_current + 1;
END WHILE;
p_sum = v_current;
END smn;
```

Note: Logic such as is shown in the example above would be better implemented using a mathematical formula. The simple example serves to demonstrate the syntax.

*REPEAT statement in SQL procedures:* The REPEAT statement defines a set of statements to be executed until a condition that is evaluated at the end of the REPEAT loop is true. The repeat-loop-condition is evaluated at the completion of each iteration of the loop.

With a WHILE statement, the loop is not entered if the while-loop-condition is false at 1st pass. The REPEAT statement is useful alternative; however it is noteworthy that while-loop logic can be rewritten as a REPEAT statement.

Here is an SQL procedure that includes a REPEAT statement:

```
CREATE PROCEDURE sum_mn2 (IN p_start INT
                         ,IN p_end INT
                         ,OUT p_sum INT)
SPECIFIC sum_mn2
LANGUAGE SQL
smn2: BEGIN

  DECLARE v_temp INTEGER DEFAULT 0;
  DECLARE v_current INTEGER;

  SET v_current = p_start;

  REPEAT
    SET v_temp = v_temp + v_current;
    SET v_current = v_current + 1;
  UNTIL (v_current > p_end)
  END REPEAT;
END
```

*Transfer of control statements in SQL procedures:* Transfer of control statements are used to redirect the flow of control within an SQL procedure. This unconditional branching can be used to cause the flow of control to jump from one point to another point, which can either precede or follow the transfer of control statement. The supported transfer of control statements in SQL procedures are:

- GOTO
- ITERATE
- LEAVE
- RETURN

Transfer of control statements can be used anywhere within an SQL procedure, however ITERATE and LEAVE are generally used in conjunction with a LOOP statement or other looping statements.

*GOTO statement in SQL procedures:* The GOTO statement is a straightforward and basic flow of control statement that causes an unconditional change in the flow of control. It is used to branch to a specific user-defined location using labels defined in the SQL procedure.

Use of the GOTO statement is generally considered to be poor programming practice and is not recommended. Extensive use of GOTO tends to lead to unreadable code especially when procedures grow long. Besides, GOTO is not necessary because there are better statements available to control the execution path. There are no specific situations that require the use of GOTO; instead it is more often used for convenience.

Here is an example of an SQL procedure that contains a GOTO statement:

```
CREATE PROCEDURE adjust_salary ( IN p_empno CHAR(6),
                     IN p_rating INTEGER,
     OUT p_adjusted_salary DECIMAL (8,2) )
LANGUAGE SQL
BEGIN
  DECLARE new_salary DECIMAL (9,2);
  DECLARE service DATE;  -- start date

  SELECT salary, hiredate INTO v_new_salary, v_service
    FROM employee
       WHERE empno = p_empno;

  IF service > (CURRENT DATE - 1 year) THEN
    GOTO exit;
  END IF;

  IF p_rating = 1 THEN
    SET new_salary = new_salary + (new_salary * .10);
  END IF;

  UPDATE employee SET salary = new_salary WHERE empno = p_empno;

exit:
  SET p_adjusted_salary = v_new_salary;


END
```

This example demonstrates what of the good uses of the GOTO statement: skipping almost to the end of a procedure or loop so as not to execute some logic, but to ensure that some other logic does still get executed.

You should be aware of a few additional scope considerations when using the GOTO statement:

- If the GOTO statement is defined in a FOR statement, the label must be defined inside the same FOR statement, unless it is in a nested FOR statement or nested compound statement.
- If the GOTO statement is defined in a compound statement, the label must be defined in side the same compound statement, unless it is in a nested FOR statement or nested compound statement.
- If the GOTO statement is defined in a handler, the label must be defined in the same handler, following the other scope rules.
- If the GOTO statement is defined outside of a handler, the label must not be defined within a handler.
- If the label is not defined within a scope that the GOTO statement can reach, an error is returned (SQLSTATE 42736).

*ITERATE statement in SQL procedures:* The ITERATE statement is used to cause the flow of control to return to the beginning of a labeled LOOP statement.

Here is an example of an SQL procedure that contains an ITERATE statement:

```
CREATE  PROCEDURE ITERATOR()
LANGUAGE SQL
BEGIN
  DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
  DECLARE at_end INTEGER DEFAULT 0;
  DECLARE not_found CONDITION FOR SQLSTATE '02000';

  DECLARE c1 CURSOR FOR SELECT deptno, deptname
```

```
                          FROM department ORDER BY deptno;
    DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;
    OPEN c1;

    ins_loop: LOOP
     FETCH c1 INTO v_deptno, v_deptname;
        IF at_end = 1 THEN
          LEAVE ins_loop;
        ELSEIF v_dept = 'D11' THEN
          ITERATE ins_loop;
        END  IF;

        INSERT  INTO department (deptno, deptname)
        VALUES ('NEW', v_deptname);

    END LOOP;

    CLOSE c1;

  END
```

In the example, the ITERATE statement is used to return the flow of control to the LOOP statement defined with label ins_loop when a column value in a fetched row matches a certain value. The position of the ITERATE statement ensures that no values are inserted into the department table.

*LEAVE statement in SQL procedures:* The LEAVE statement is used to transfer the flow of control out of a loop or compound statement.

Here is an example of an SQL procedure that contain a LEAVE statement:

```
  CREATE  PROCEDURE ITERATOR()
  LANGUAGE SQL
  BEGIN
    DECLARE v_deptno CHAR(3); DECLARE v_deptname VARCHAR(29);
    DECLARE at_end INTEGER DEFAULT 0;
    DECLARE not_found CONDITION FOR SQLSTATE '02000';

    DECLARE c1 CURSOR FOR SELECT deptno, deptname
                          FROM department ORDER BY deptno;
    DECLARE CONTINUE HANDLER FOR not_found SET at_end = 1;

    OPEN c1;

    ins_loop: LOOP

      FETCH c1 INTO v_deptno, v_deptname;

      IF at_end = 1 THEN
        LEAVE ins_loop;
      ELSEIF v_dept = 'D11' THEN
        ITERATE ins_loop;
      END IF;

      INSERT  INTO department (deptno, deptname)
      VALUES ('NEW', v_deptname);

    END LOOP;

    CLOSE c1;
END
```

In the example, the LEAVE statement is used to exit the LOOP statement defined with label ins_loop. It is nested within an IF statement and therefore is conditionally executed when the IF-condition is true which becomes true when there are no more rows found in the cursor. The position of the LEAVE statement ensures that no further iterations of the loop are executed once a NOT FOUND error is raised.

*RETURN statement in SQL procedures:* The RETURN statement is used to unconditionally and immediately terminate an SQL procedure by returning the flow of control to the caller of the stored procedure.

It is mandatory that when the RETURN statement is executed that it return an integer value. If the return value is not provided, the default is 0. The value is typically used to indicate success or failure of the procedure's execution. The value can be a literal, variable, or an expression that evaluates to an integer value.

You can use one or more RETURN statements in a stored procedure. The RETURN statement can be used anywhere after the declaration blocks within the SQL-procedure-body.

To return multiple output values, parameters can be used instead. Parameter values must be set prior to the RETURN statement being executed.

Here is an example of an SQL procedure that uses the RETURN statement:

```
CREATE PROCEDURE return_test (IN p_empno CHAR(6),
                              IN p_emplastname VARCHAR(15) )
LANGUAGE SQL
SPECIFIC return_test
BEGIN

  DECLARE v_lastname VARCHAR (15);

  SELECT lastname INTO v_lastname
    FROM employee
  WHERE empno = p_empno;

  IF v_lastname = p_emplastname THEN
    RETURN 1;
  ELSE
    RETURN -1;
  END IF;

END rt
```

In the example, if the parameter *p_emplastname* matches the value stored in table employee, the procedure returns 1. If it does not match, it returns -1.

*Condition handlers in SQL procedures:* Condition handlers determine the behavior of your SQL procedure when a condition occurs. You can declare one or more condition handlers in your SQL procedure for general conditions, named conditions, or specific SQLSTATE values.

If a statement in your SQL procedure raises an SQLWARNING or NOT FOUND condition, and you have declared a handler for the respective condition, DB2 passes control to the corresponding handler. If you have not declared a handler for such a condition, DB2 passes control to the next statement in the SQL procedure body. If the SQLCODE and SQLSTATE variables have been declared, they will contain the corresponding values for the condition.

If a statement in your SQL procedure raises an SQLEXCEPTION condition, and you declared a handler for the specific SQLSTATE or the SQLEXCEPTION condition, DB2 passes control to that handler. If the SQLSTATE and SQLCODE variables have been declared, their values after the successful execution of a handler will be ‘00000' and 0 respectively.

If a statement in your SQL procedure raises an SQLEXCEPTION condition, and you have not declared a handler for the specific SQLSTATE or the SQLEXCEPTION condition, DB2 terminates the SQL procedure and returns to the caller.

The handler declaration syntax for condition handlers is described in Compound SQL (Procedure) statement.

**Returning result sets from SQL procedures:**

In SQL procedures, cursors can be used to do more than iterate through rows of a result set. They can also be used to return result sets to the calling program. Result sets can be retrieved by SQL procedures (in the case of a nested procedure calls) or client applications programmed in C using the CLI application programming interface, Java, CLI, or .NET CLR languages.

**Prerequisites**

- Authority to create an SQL procedure

To return a result set from an SQL procedure, you must:

1. Specify the DYNAMIC RESULT SETS clause in the CREATE PROCEDURE statement
2. DECLARE the cursor using the WITH RETURN clause
3. Open the cursor in the SQL procedure
4. Keep the cursor open for the client application - do not close it

Here is an example of an SQL procedure that only returns a single result set:

```
CREATE PROCEDURE  read_emp()
SPECIFIC read_emp
LANGUAGE SQL
DYNAMIC RESULT SETS 1

Re:  BEGIN

  DECLARE c_emp CURSOR WITH RETURN FOR
    SELECT salary, bonus, comm.
    FROM employee
    WHERE job != 'PRES';

  OPEN c_emp;

END Re
```

If the cursor is closed using the CLOSE statement prior to the return of the SQL procedure, the cursor result set will not be returned to the caller or client application.

Multiple result sets can be returned from an SQL procedure by using multiple cursors. To return multiple cursors the following must be done:

- Specify the DYNAMIC RESULT SETS clause in the CREATE PROCEDURE statement. Specify the maximum possible number of result sets likely to be returned. The number of results sets actually returned must not exceed this number.

- Declare cursors for each of the result sets to be returned that specify the WITH RETURN clause.
- Open the cursors to be returned.
- Keep the cursor open for the client application - do not close them.

One cursor is required per result set that is to be returned.

Result sets are returned to the caller in the order in which they are opened.

Once you have created the SQL procedure that returns a result set you might want to call it and retrieve the result set.

Multiple result sets can also be returned by enabling multiple instances of a same cursor. You must DECLARE the cursor using the WITH RETURN TO CLIENT.

An example to enable multiple instances of an open cursor using the WITH RETURN TO CLIENT:

```
CREATE PROCEDURE PROC(IN a INT)
BEGIN
  DECLARE index INTEGER DEFAULT 1;
  WHILE index < a DO
    BEGIN
      DECLARE cur CURSOR WITH RETURN TO CLIENT FOR SELECT * FROM T WHERE pk = index;
      OPEN cur;
      SET index = index + 1;
    END;
  END WHILE;
END
@
```

**Receiving procedure result sets in SQL routines:**

You can receive result sets from procedures you invoke from within an SQL-bodied routine.

You must know how many result sets the invoked procedure will return. For each result set that the invoking routine receives, a result set must be declared.

To accept procedure result sets from within an SQL-bodied routine:

1. DECLARE result set locators for each result set that the procedure will return. For example:
   ```
   DECLARE result1 RESULT_SET_LOCATOR VARYING;
   DECLARE result2 RESULT_SET_LOCATOR VARYING;
   DECLARE result3 RESULT_SET_LOCATOR VARYING;
   ```
2. Invoke the procedure. For example:
   ```
   CALL targetProcedure();
   ```
3. ASSOCIATE the result set locator variables (defined above) with the invoked procedure. For example:
   ```
   ASSOCIATE RESULT SET LOCATORS(result1, result2, result3)
      WITH PROCEDURE targetProcedure;
   ```
4. ALLOCATE the result set cursors passed from the invoked procedure to the result set locators. For example:
   ```
   ALLOCATE rsCur CURSOR FOR RESULT SET result1;
   ```
5. FETCH rows from the result sets. For example:
   ```
   FETCH rsCur INTO ...
   ```

## Creating SQL procedures

Creating SQL procedures is similar to creating any database object in that it consists of executing a DDL SQL statement.

SQL procedures are created by executing the CREATE PROCEDURE statement which can be done using graphical development environment tools or by directly executing the statement from the DB2 Command Line Processor (CLP), a DB2 Command Window, the DB2 Command Editor, or another DB2 interface.

When creating SQL procedures, you can specify how the precompiler and binder should generate the procedure package, what authorization ID should be used to set the SQL procedure definer in the DB2 catalog views, and to set other package options.

**Creating SQL procedures from the command line:**

- The user must have the privileges required to execute the CREATE PROCEDURE statement for an SQL procedure.
- Privileges to execute all of the SQL statements included within the SQL-procedure-body of the procedure.
- Any database objects referenced in the CREATE PROCEDURE statement for the SQL procedure must exist prior to the execution of the statement.
- Select an alternate terminating character for the Command Line Processor (DB2 CLP) other than the default terminating character, which is a semicolon (';'), to use in the script that you will prepare in the next step.

  This is required so that the CLP can distinguish the end of SQL statements that appear within the body of a routine's CREATE statement from the end of the CREATE PROCEDURE statement itself. The semicolon character must be used to terminate SQL statements within the SQL routine body and the chosen alternate terminating character should be used to terminate the CREATE statement and any other SQL statements that you might contain within your CLP script.

  For example, in the following CREATE PROCEDURE statement, the 'at;' sign ('@') is used as the terminating character for a DB2 CLP script named `myCLPscript.db2`:

```
    CREATE PROCEDURE UPDATE_SALARY_IF
    (IN employee_number CHAR(6), IN rating SMALLINT)
    LANGUAGE SQL
    BEGIN
      DECLARE not_found CONDITION FOR SQLSTATE '02000';
      DECLARE EXIT HANDLER FOR not_found
         SIGNAL SQLSTATE '20000' SET MESSAGE_TEXT = 'Employee not found';

      IF (rating = 1)
        THEN UPDATE employee
          SET salary = salary * 1.10, bonus = 1000
          WHERE empno = employee_number;
      ELSEIF (rating = 2)
        THEN UPDATE employee
          SET salary = salary * 1.05, bonus = 500
          WHERE empno = employee_number;
      ELSE UPDATE employee
          SET salary = salary * 1.03, bonus = 0
          WHERE empno = employee_number;
      END IF;
    END
@
```

- Run the DB2 CLP script containing the CREATE PROCEDURE statement for the procedure from the command line, using the following CLP command:

```
db2 -td terminating-character -vf CLP-script-name
```

where *terminating-character* is the terminating character used in the CLP script file *CLP-script-name* that is to be run.

The DB2 CLP option **-td** indicates that the CLP terminator default is to be reset with *terminating-character*. The **-vf** indicates that the CLP's optional verbose (**-v**) option is to be used, which will cause each SQL statement or command in the script to be displayed to the screen as it is run, along with any output that results from its execution. The **-f** option indicates that the target of the command is a file.

To run the specific script shown in the first step, issue the following command from the system command prompt:

```
db2 -td@ -vf myCLPscript.db2
```

**Customizing precompile and bind options for compiled SQL objects:**

The precompile and bind options for SQL procedures, compiled functions, compiled triggers and compound SQL (complied) statements can be customized by setting the instance-wide DB2 registry variable, `DB2_SQLROUTINE_PREPOPTS` with the command:

```
db2set DB2_SQLROUTINE_PREPOPTS=<options>
```

The options can be changed at the procedure level with the `SET_ROUTINE_OPTS` stored procedure. The values of the options set for the creation of SQL procedures in the current session can be obtained with the `GET_ROUTINE_OPTS` function.
The options used to compile a given routine are stored in the system catalog table ROUTINES.PRECOMPILE_OPTIONS, in the row corresponding to the routine. If the routine is revalidated, those stored options are also used during the revalidation.
After a routine is created, the compile options can be altered using the SYSPROC.ALTER_ROUTINE_PACKAGE and SYSPROC.REBIND_ROUTINE_PACKAGE procedures. The altered options are reflected in the ROUTINES_PRECOMPILE_OPTIONS system catalog table.

**Note:** Cursor blocking is disabled in SQL procedures for cursors referenced in FETCH statements and for implicit cursors in FOR statements. Regardless of the value specified for the BLOCKING bind option, data will be retrieved one row at a time in an optimized, highly efficient manner.

**Example.**

The SQL procedures used in this example will be defined in CLP scripts (given below). These scripts are not in the `sqlpl` samples directory, but you can easily create these files by cutting-and-pasting the CREATE procedure statements into your own files.

The examples use a table named "expenses", which you can create in the sample database as follows:

```
db2 connect to sample
db2 CREATE TABLE expenses(amount DOUBLE, date DATE)
db2 connect reset
```

To begin, specify the use of ISO format for dates as an instance-wide setting:

```
db2set DB2_SQLROUTINE_PREPOPTS="DATETIME ISO"
db2stop
db2start
```

Stopping and restarting DB2 is necessary for the change to take affect.

Then connect to the database:

```
   db2 connect to sample
```

The first procedure is defined in CLP script maxamount.db2 as follows:

```
   CREATE PROCEDURE maxamount(OUT maxamnt DOUBLE)
   BEGIN
     SELECT max(amount) INTO maxamnt FROM expenses;
   END @
```

It will be created with options DATETIME ISO and ISOLATION UR:

```
   db2 "CALL SET_ROUTINE_OPTS(GET_ROUTINE_OPTS() || ' ISOLATION UR')"
   db2 -td@ -vf maxamount.db2
```

The next procedure is defined in CLP script fullamount.db2 as follows:

```
CREATE PROCEDURE fullamount(OUT fullamnt DOUBLE)
BEGIN
  SELECT sum(amount) INTO fullamnt FROM expenses;
END @
```

It will be created with option ISOLATION CS (note that we are not using the instance-wide DATETIME ISO setting in this case):

```
   CALL SET_ROUTINE_OPTS('ISOLATION CS')
   db2 -td@ -vf fullamount.db2
```

The last procedure in the example is defined in CLP script perday.db2 as follows:

```
CREATE PROCEDURE perday()
BEGIN
  DECLARE cur1 CURSOR WITH RETURN FOR
    SELECT date, sum(amount)
    FROM expenses
    GROUP BY date;

    OPEN cur1;
END @
```

The last SET_ROUTINE_OPTS call uses the NULL value as the argument. This restores the global setting specified in the DB2_SQLROUTINE_PREPOPTS registry, so the last procedure will be created with option DATETIME ISO:

```
   CALL SET_ROUTINE_OPTS(NULL)
   db2 -td@ -vf perday.db2
```

## Improving the performance of SQL procedures

### Overview of how DB2 compiles SQL PL and inline SQL PL

Before discussing how to improve the performance of SQL procedures we should discuss how DB2 compiles them upon the execution of the CREATE PROCEDURE statement.

When an SQL procedure is created, DB2 separates the SQL queries in the procedure body from the procedural logic. To maximize performance, the SQL queries are statically compiled into sections in a package. For a statically compiled query, a section consists mainly of the access plan selected by the DB2 optimizer for that query. A package is a collection of sections. For more information on packages and sections, please refer to the DB2 SQL Reference. The procedural logic is compiled into a dynamically linked library.

During the execution of a procedure, every time control flows from the procedural logic to an SQL statement, there is a "context switch" between the DLL and the DB2 engine. As of DB2 Version 8.1, SQL procedures run in "unfenced mode". That is they run in the same addressing space as the DB2 engine. Therefore the context switch we refer to here is not a full context switch at the operating system level, but rather a change of layer within DB2. Reducing the number of context switches in procedures that are invoked very often, such as procedures in an OLTP application, or that process large numbers of rows, such as procedures that perform data cleansing, can have a noticeable impact on their performance.

Whereas an SQL procedure containing SQL PL is implemented by statically compiling its individual SQL queries into sections in a package, an inline SQL PL function is implemented, as the name suggests, by inlining the body of the function into the query that uses it. Queries in SQL functions are compiled together, as if the function body were a single query. The compilation occurs every time a statement that uses the function is compiled. Unlike what happens in SQL procedures, procedural statements in SQL functions are not executed in a different layer than dataflow statements. Therefore, there is no context switch every time control flows from a procedural to a dataflow statement or vice versa.

### If there are no side-effects in your logic use an SQL function instead

Because of the difference in compilation between SQL PL in procedures and inline SQL PL in functions, it is reasonable to presume that a piece of procedural code will execute faster in a function than in a procedure if it only queries SQL data and does no data modifications - that is it has no side-effects on the data in the database or external to the database.

That is only good news if all the statements that you need to execute are supported in SQL functions. SQL functions can not contain SQL statements that modify the database. As well, only a subset of SQL PL is available in the inline SQL PL of functions. For example, you cannot execute CALL statements, declare cursors, or return result sets in SQL functions.

Here is an example of an SQL procedure containing SQL PL that was a good candidate for conversion to an SQL function to maximize performance:

```
CREATE PROCEDURE GetPrice (IN Vendor CHAR&(20&),
                           IN Pid INT, OUT price DECIMAL(10,3))
LANGUAGE SQL
BEGIN
  IF Vendor eq; ssq;Vendor 1ssq;
    THEN SET price eq; (SELECT ProdPrice
                            FROM V1Table
                            WHERE Id = Pid);
  ELSE IF Vendor eq; ssq;Vendor 2ssq;
    THEN SET price eq; (SELECT Price FROM V2Table
                            WHERE Pid eq; GetPrice.Pid);
  END IF;
END
```

Here is the rewritten SQL function:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
RETURNS  DECIMAL(10,3)
LANGUAGE SQL
BEGIN
  DECLARE price DECIMAL(10,3);
  IF Vendor = 'Vendor 1'
    THEN SET price = (SELECT ProdPrice
                             FROM V1Table
```

```
                           WHERE Id = Pid);
    ELSE IF Vendor = 'Vendor 2'
      THEN SET price = (SELECT Price FROM V2Table
                 WHERE Pid = GetPrice.Pid);
    END IF;
    RETURN price;
  END
```

Remember that the invocation of a function is different than a procedure. To invoke the function, use the VALUES statement or invoke the function where an expression is valid, such as in a SELECT or SET statement. Any of the following are valid ways of invoking the new function:

```
VALUES (GetPrice('IBM', 324))

SELECT VName FROM Vendors WHERE GetPrice(Vname, Pid) < 10

SET price = GetPrice(Vname, Pid)
```

### Avoid multiple statements in an SQL PL procedure when just one is sufficient

Although it is generally a good idea to write concise SQL, it is very ease to forget to do this in practice. For example the following SQL statements:

```
INSERT INTO tab_comp VALUES (item1, price1, qty1);
INSERT INTO tab_comp VALUES (item2, price2, qty2);
INSERT INTO tab_comp VALUES (item3, price3, qty3);
```

can be rewritten as a single statement:

```
INSERT INTO tab_comp VALUES (item1, price1, qty1),
                            (item2, price2, qty2),
                            (item3, price3, qty3);
```

The multi-row insert will require roughly one third of the time required to execute the three original statements. Isolated, this improvement might seem negligible, but if the code fragment is executed repeatedly, for example in a loop or in a trigger body, the improvement can be significant.

Similarly, a sequence of SET statements like:

```
SET A = expr1;
SET B = expr2;
SET C = expr3;
```

can be written as a single VALUES statement:

```
VALUES expr1, expr2, expr3 INTO A, B, C;
```

This transformation preserves the semantics of the original sequence if there are no dependencies between any two statements. To illustrate this, consider:

```
SET A = monthly_avg * 12;
SET B = (A / 2) * correction_factor;
```

Converting the previous two statements to:

```
VALUES (monthly_avg * 12, (A / 2) * correction_factor) INTO A, B;
```

does not preserve the original semantics because the expressions before the INTO keyword are evaluated 'in parallel'. This means that the value assigned to *B* is not based on the value assigned to *A*, which was the intended semantics of the original statements.

## Reduce multiple SQL statements to a single SQL expression

Like other programming languages, the SQL language provides two types of conditional constructs: procedural (IF and CASE statements) and functional (CASE expressions). In most circumstances where either type can be used to express a computation, using one or the other is a matter of taste. However, logic written using CASE expressions is not only more compact, but also more efficient than logic written using CASE or IF statements.

Consider the following fragment of SQL PL code:

```
IF (Price <= MaxPrice) THEN
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, Price);
ELSE
  INSERT INTO tab_comp(Id, Val) VALUES(Oid, MaxPrice);
END IF;
```

The condition in the IF clause is only being used to decide what value is inserted in the tab_comp.Val column. To avoid the context switch between the procedural and the dataflow layers, the same logic can be expressed as a single INSERT with a CASE expression:

```
INSERT INTO tab_comp(Id, Val)
       VALUES(Oid,
            CASE
               WHEN (Price <= MaxPrice) THEN Price
               ELSE MaxPrice
            END);
```

It's worth noting that CASE expressions can be used in any context where a scalar value is expected. In particular, they can be used on the right-hand side of assignments. For example:

```
IF (Name IS NOT NULL) THEN
  SET ProdName = Name;
ELSEIF (NameStr IS NOT NULL) THEN
  SET ProdName = NameStr;
ELSE
  SET ProdName = DefaultName;
END IF;
```

can be rewritten as:

```
SET ProdName = (CASE
                  WHEN (Name IS NOT NULL) THEN Name
                  WHEN (NameStr IS NOT NULL) THEN NameStr
                  ELSE  DefaultName
                 END);
```

In fact, this particular example admits an even better solution:

```
SET ProdName = COALESCE(Name, NameStr, DefaultName);
```

Don't underestimate the benefit of taking the time to analyze and consider rewriting your SQL. The performance benefits will pay you back many times over for the time invested in analyzing and rewriting your procedure.

## Exploit the set-at-a-time semantics of SQL

Procedural constructs such as loops, assignment and cursors allow us to express computations that would not be possible to express using just SQL DML statements. But when we have procedural statements at our disposal, there is a risk that we could turn to them even when the computation at hand can, in fact, be expressed using just SQL DML statements. As we've mentioned earlier, the performance of a procedural computation can be orders of magnitude slower than the performance of an equivalent computation expressed using DML statements. Consider the following fragment of code:

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_sel VALUES (20, v2);
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

To begin with, the loop body can be improved by applying the transformation discussed in the last section - "Reduce multiple SQL statements to a single SQL expression":

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  INSERT INTO tab_sel VALUES (CASE
                                WHEN v1 > 20 THEN 20
                                ELSE v1
                              END, v2);
  FETCH cur1 INTO v1, v2;
END WHILE;
```

But upon closer inspection, the whole block of code can be written as an INSERT with a sub-SELECT:

```
INSERT INTO tab_sel (SELECT (CASE
                               WHEN col1 > 20 THEN 20
                               ELSE col1
                             END),
                             col2
                     FROM tab_comp);
```

In the original formulation, there was a context switch between the procedural and the dataflow layers for each row in the SELECT statements. In the last formulation, there is no context switch at all, and the optimizer has a chance to globally optimize the full computation.

On the other hand, this dramatic simplification would not have been possible if each of the INSERT statements targeted a different table, as shown below:

```
DECLARE cur1 CURSOR FOR SELECT col1, col2 FROM tab_comp;
OPEN cur1;
FETCH cur1 INTO v1, v2;
WHILE SQLCODE <> 100 DO
  IF (v1 > 20) THEN
    INSERT INTO tab_default VALUES (20, v2);
  ELSE
    INSERT INTO tab_sel VALUES (v1, v2);
  END IF;
  FETCH cur1 INTO v1, v2;
END WHILE;
```

However, the set-at-a-time nature of SQL can also be exploited here:

```
INSERT INTO tab_sel (SELECT col1, col2
                     FROM tab_comp
                     WHERE col1 <= 20);
INSERT INTO tab_default (SELECT col1, col2
                         FROM tab_comp
                         WHERE col1 > 20);
```

When looking at improving the performance of existing procedural logic, any time spent in eliminating cursor loops will likely pay off.

### Keep the DB2 optimizer informed

When a procedure is created, its individual SQL queries are compiled into sections in a package. The DB2 optimizer chooses an execution plan for a query based, among other things, on table statistics (for example, table sizes or the relative frequency of data values in a column) and indexes available at the time the query is compiled. When tables suffer significant changes, it may be a good idea to let DB2 collect statistics on these tables again. And when statistics are updated or new indexes are created, it may also be a good idea to rebind the packages associated with SQL procedures that use the tables, to let DB2 create plans that exploit the latest statistics and indexes.

Table statistics can be updated using the RUNSTATS command. To rebind the package associated with an SQL procedure, you can use the REBIND_ROUTINE_PACKAGE built-in procedure that is available in DB2 Version 8.1. For example, the following command can be used to rebind the package for procedure MYSCHEMA.MYPROC:

```
CALL SYSPROC.REBIND_ROUTINE_PACKAGE('P', 'MYSCHEMA.MYPROC', 'ANY')
```

where 'P' indicates that the package corresponds to a procedure and 'ANY' indicates that any of the functions and types in the SQL path are considered for function and type resolution. See the Command Reference entry for the REBIND command for more details.

### Use arrays

You can use arrays to efficiently pass collections of data between applications and stored procedures and to store and manipulate transient collections of data within SQL procedures without having to use relational tables. Operators on arrays available within SQL procedures allow for the efficient storage and retrieval of data. Applications that create arrays of moderate size will experience significantly better performance than applications that create very large arrays (on the scale of multiple megabytes), as the entire array is stored in main memory. See *Related links* section for additional information.

## SQL functions

SQL functions are functions implemented completely with SQL that can be used to encapsulate logic that can be invoked like a programming sub-routine. You can create SQL scalar functions and SQL table functions.

There are many useful applications for SQL functions within a database or database application architecture. SQL functions can be used to create operators on column data, for extending the support of built-in functions, for making application logic more modular, and for improving overall database design, and database security.

The following topics provide more information about SQL functions:

## Features of SQL functions

SQL functions are characterized by many general features:

SQL functions:

- Can contain SQL Procedural Language statements and features which support the implementation of control-flow logic around traditional static and dynamic SQL statements.
- Are supported in the entire DB2 family brand of database products in which many if not all of the features supported in DB2 Version 9 are supported.
- Are easy to implement, because they use a simple high-level, strongly typed language.
- SQL functions are more reliable than equivalent external functions.
- Support input parameters.
- SQL scalar functions return a scalar value.
- SQL table functions return a table result set.
- Support a simple, but powerful condition and error-handling model.
- Allow you to easily access the SQLSTATE and SQLCODE values as special variables.
- Reside in the database and are automatically backed up and restored as part of backup and restore operations.
- Can be invoked wherever expressions in an SQL statement are supported.
- Support nested functions calls to other SQL functions or functions implemented in other languages.
- Support recursion (when dynamic SQL is used in compiled functions).
- Can be invoked from triggers.
- Many SQL statements can be included within SQL functions, however there are exceptions. For the complete list of SQL statements that can included and executed in SQL functions, see:SQL statements that can be executed in routines

SQL functions provide extensive support not limited to what is listed above. When implemented according to best practices, they can play an essential role in database architecture, database application design, and in database system performance.

## Designing SQL functions

Designing SQL functions is a task that you perform before creating SQL functions in a database.

To design SQL functions it is important to be familiar with the features of SQL functions. The following topics provide more information about SQL function design concepts:

### Inlined SQL functions and compiled SQL functions:

There are two implementation types for SQL functions: inlined SQL functions and compiled SQL functions.

**Inlined SQL functions**

Inlined SQL functions are SQL functions that are created using the CREATE FUNCTION statement with a body that is either a RETURN statement or an inline compound statement. Inline compound statements are defined with the BEGIN ATOMIC and END keywords.

Inlined SQL functions can contain SQL statements and inline SQL PL statements - a subset of SQL PL statements.

**Compiled SQL functions**

Compiled SQL functions are SQL functions that are created using the CREATE FUNCTION statement with a body that is either a RETURN statement or a compiled compound statement. Compiled compound statements are defined with the BEGIN and END keywords.

When the ATOMIC clause is omitted, SQL functions are compiled and as such can include or reference more SQL PL features than inlined SQL functions. Compiled SQL functions can include the following features which are not supported in inlined SQL functions:

- SQL PL statements, including:
  - CASE statement
  - REPEAT statement
- Cursor processing
- Dynamic SQL
- Condition handlers

**Restrictions on SQL functions:**

It is important to note the restrictions on SQL functions before creating them or when troubleshooting problems related to their implementation and use.

The following restrictions apply to SQL functions:

- SQL table functions cannot contain compiled compound statements.
- SQL scalar functions containing compiled compound statements cannot be invoked in partitioned database environments.
- By definition, SQL functions cannot contain cursors defined with the WITH RETURN clause.
- Compiled SQL scalar functions cannot be invoked in partitioned database environments.
- The following data types are not supported within compiled SQL functions: structured data types, XML data type, LONG VARCHAR data type, and LONG VARGRAPHIC data type.
- In this version, use of the DECLARE TYPE statement within compiled SQL functions is not supported.

## Creating SQL scalar functions

Creating SQL scalar functions is a task that you would perform when designing a database or when developing applications. SOL scalar functions are generally created when there is an identifiable benefit in encapsulating a piece of reusable logic so that it can be referenced within SQL statements in multiple applications or within database objects.

Before you create an SQL function:

- Read: “SQL functions” on page 89
- Read: “Features of SQL functions” on page 90

- Ensure that you have the privileges required to execute the CREATE FUNCTION (scalar) statement.

**Restrictions**

See:"Restrictions on SQL functions" on page 91

1. Define the CREATE FUNCTION (scalar) statement:
   a. Specify a name for the function.
   b. Specify a name and data type for each input parameter.
   c. Specify the RETURNS keyword and the data type of the scalar return value.
   d. Specify the BEGIN keyword to introduce the function-body. Note: Use of the BEGIN ATOMIC keyword is not recommended for new functions.
   e. Specify the function body. Specify the RETURN clause and a scalar return value or variable.
   f. Specify the END keyword.
2. Execute the CREATE FUNCTION (scalar) statement from a supported interface.

The CREATE FUNCTION (scalar) statement should execute successfully and the scalar function should be created.

**Example 1:**

The following is an example of a compiled SQL function:

```
CREATE FUNCTION GetPrice (Vendor CHAR(20), Pid INT)
  RETURNS  DECIMAL(10,3)
  LANGUAGE SQL
  MODIFIES SQL
  BEGIN
    DECLARE price DECIMAL(10,3);

    IF Vendor = 'Vendor 1'
      THEN SET price = (SELECT ProdPrice FROM V1Table WHERE Id = Pid);
    ELSE IF Vendor = 'Vendor 2'
      THEN SET price = (SELECT Price
                        FROM V2Table
           WHERE Pid = GetPrice.Pid);
    END IF;

  RETURN price;
END
```

This function takes in two input parameters and returns a single scalar value, conditionally based on the input parameter values. It requires the declaration and use of a local variable named price to hold the value to be returned until the function returns.

**Example 2:**

The following example demonstrates a compiled SQL function definition containing a cursor, condition handler statement, and a REPEAT statement:

```
CREATE FUNCTION exit_func(a INTEGER)
  SPECIFIC exit_func
  LANGUAGE SQL
  RETURNS INTEGER
  BEGIN
    DECLARE val INTEGER DEFAULT 0;
```

```
      DECLARE myint INTEGER DEFAULT 0;

      DECLARE cur2 CURSOR FOR
        SELECT c2 FROM udfd1
          WHERE c1 <= a
          ORDER BY c1;

      DECLARE EXIT HANDLER FOR NOT FOUND
        BEGIN
          SIGNAL SQLSTATE '70001'
          SET MESSAGE_TEXT =
            'Exit handler for not found fired';
        END;

    OPEN cur2;

    REPEAT
      FETCH cur2 INTO val;
      SET myint = myint + val;
    UNTIL (myint >= a)
    END REPEAT;

    CLOSE cur2;

    RETURN myint;

  END@
```

After creating the scalar function you might want to invoke the function to test it.

## Creating SQL table functions

The task of creating SQL table functions can be done at any time.

Before you create an SQL table function ensure that you have the privileges required to execute the CREATE FUNCTION (table) statement.

**Restrictions**

See: "Restrictions on SQL functions" on page 91

1. Define the CREATE FUNCTION (table) statement:
   a. Specify a name for the function.
   b. Specify a name and data type for each input parameter.
   c. Specify the routine attributes.
   d. Specify the RETURNS TABLE keyword.
   e. Specify the BEGIN ATOMIC keyword to introduce the function-body.
   f. Specify the function body.
   g. Specify the RETURN clause with brackets in which you specify a query that defines the result set to be returned.
   h. Specify the END keyword.
2. Execute the CREATE FUNCTION (table) statement from a supported interface.

The CREATE FUNCTION (table) statement should execute successfully and the table function should be created.

**Example 1:**

The following is an example of a compiled SQL table function that is used to track and audit updates made to employee salary data:

```
CREATE FUNCTION update_salary  (updEmpNum CHAR(4), amount INTEGER)
RETURNS TABLE (emp_lastname VARCHAR(10),
      emp_firstname VARCHAR(10),
      newSalary INTEGER)
LANGUAGE SQL
MODIFIES SQL DATA
NO EXTERNAL ACTION
NOT DETERMINISTIC
BEGIN ATOMIC

  INSERT INTO audit_table(user, table, action, time)
    VALUES (USER, 'EMPLOYEE',
    'Salary update. Values: ' || updEmpNum || ' ' || char(amount), CURRENT_TIMESTAMP);

  RETURN (SELECT lastname, firstname, salary
    FROM FINAL TABLE(UPDATE employee SET salary = salary + amount WHERE employee.empnum = updEmpNum));

END
```

This function updates the salary of an employee specified by updEmpNum, by the amount specified by amount, and also records in an audit table named audit_table, the user that invoked the routine, the name of the table that was modified, and the type of modification made by the user. A SELECT statement that references a data change statement in the FROM clause is used to get back the updated row values.

**Example 2:**

The following is an example of a compiled SQL table function:

```
CREATE TABLE t1(pk INT

CREATE TABLE t1_archive LIKE T1%

CREATE FUNCTION archive_tbl_t1(ppk INT)
  RETURNS TABLE(pk INT, c1 INT, date)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN ATOMIC

  DECLARE c1 INT;

  DECLARE date DATE;

  SET (c1, date) = (SELECT * FROM OLD TABLE(DELETE FROM t1 WHERE t1.pk = ppk));

  INSERT INTO T1_ARCHIVE VALUES (ppk, c1, date);

  RETURN VALUES (pk, c1, date);
END%
```

After creating the table function you might want to invoke the function to test it.

## Compound statements

A compound statement groups other statements together into an executable block. Compound statements can be executed independently or be included in the definitions of database objects such as procedures, functions, methods, and triggers. There are different SQL statements for these because there are unique differences and restrictions that apply to each.

Compound statements can be either inline compound statements (formerly called dynamic compound statements) or compiled compound statements. The differences between these two statements are shown below.

**Inline compound statements**

Inline compound statements are atomic and are defined with the BEGIN ATOMIC and END keywords, between which other SQL statements can be defined and executed. Inline compound statements can contain variable declarations, SQL statements and the subset of SQL PL statements known as inline SQL PL statements.

**Compiled compound statements**

Compiled compound statements are not atomic and are defined with the BEGIN and END keywords, between which other SQL statements can be defined and executed. Compiled compound statements can contain SQL statements and all SQL PL statements.

You would choose to use a compiled compound statement instead of an inline compound statement if you want to make use of the additional features available with compiled compound statements.

**Uses of compound statements**

Compound statements are primarily useful for creating short scripts that can be executed from the DB2 Command Line Processor. They are also used to define the body of a routine or trigger.

## Restrictions on compound statements

It is important to note the restrictions on compound statements before creating them or when troubleshooting problems related to their implementation and use.

The following restrictions apply to inlined SQL functions:

- Only a subset of SQL PL statements are supported.
- The DECLARE TYPE statement is not supported.

The following restrictions apply to compiled SQL functions:

- SQL table functions cannot contain compiled compound statements.
- SQL scalar functions containing compiled compound statements cannot be invoked in partitioned database environments.
- The DECLARE TYPE statement is supported, but the following data types are not supported with its use: structured data types, XML data type, LONG VARCHAR data type, and LONG VARGRAPHIC data type.

## Creating compound statements

Creating and executing compound statements is a task that you would perform when you need to run a script consisting of SQL statements.

Before you create a compound statement:

- Read: “Compound statements” on page 94
- Ensure that you have the privileges required to execute the Compound statement.

**Restrictions**

For a list of restrictions on compound statements, so:

- “Restrictions on compound statements”

1. Define a compound SQL statement.

2. Execute the compound SQL statement from a supported interface.

If executed dynamically, the SQL statement should execute successfully.

The following is an example of an inlined compound SQL statement that contains SQL PL:

```
BEGIN
 FOR row AS
   SELECT pk, c1, discretize(c1) AS v FROM source
 DO
   IF row.v is NULL THEN
     INSERT INTO except VALUES(row.pk, row.c1);
   ELSE
     INSERT INTO target VALUES(row.pk, row.d);
   END IF;
 END FOR;
END
```

The compound statement is bounded by the keywords BEGIN and END. It includes use of both the FOR and IF/ELSE control-statements that are part of SQL PL. The FOR statement is used to iterate through a defined set of rows. For each row a column's value is checked and conditionally, based on the value, a set of values is inserted into another table.

# Chapter 2. PL/SQL support

PL/SQL (Procedural Language/Structured Query Language) statements can be compiled and executed using DB2 interfaces. This support reduces the complexity of enabling existing PL/SQL solutions so that they will work with the DB2 data server.

The supported interfaces include:

- DB2 command line processor (CLP)
- DB2 CLPPlus
- IBM Data Studio
- IBM Optim™ Development Studio

PL/SQL statement execution is not enabled from these interfaces by default. PL/SQL statement execution support must be enabled on the DB2 data server.

## PL/SQL features

PL/SQL statements and scripts can be compiled and executed using DB2 interfaces.

You can execute the following PL/SQL statements:

- Anonymous blocks; for example, DECLARE...BEGIN...END
- CREATE OR REPLACE FUNCTION statement
- CREATE OR REPLACE PACKAGE statement
- CREATE OR REPLACE PACKAGE BODY statement
- CREATE OR REPLACE PROCEDURE statement
- CREATE OR REPLACE TRIGGER statement
- DROP PACKAGE statement
- DROP PACKAGE BODY statement

PL/SQL procedures and functions can be invoked from other PL/SQL statements or from DB2 SQL PL statements. You can call a PL/SQL procedure from SQL PL by using the CALL statement.

The following statements and language elements are supported in PL/SQL contexts:

- Type declarations (In this version, type declarations are only supported within packages. They are not supported within procedures, functions, triggers, or anonymous blocks.)
  - Associative arrays
  - Record types
  - VARRAY types
- Variable declarations:
  - %ROWTYPE
  - %TYPE
- Basic statements, clauses, and statement attributes:
  - Assignment statement

© Copyright IBM Corp. 1993, 2010

  - NULL statement
  - RETURNING INTO clause
  - Statement attributes, including SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT
- Control statements and structures:
  - CASE statements:
    - Simple CASE statement
    - Searched CASE statement
  - Exception handling
  - EXIT statement
  - FOR statement
  - GOTO statement
  - IF statement
  - LOOP statement
  - WHILE statement
- Static cursors:
  - CLOSE statement
  - Cursor FOR loop statement
  - FETCH statement (including FETCH INTO a %ROWTYPE variable)
  - OPEN statement
  - Parameterized cursors
  - Cursor attributes
- REF CURSOR support:
  - Variables and parameters of type REF CURSOR
  - Strong REF CURSORs
  - OPEN FOR statement
  - Returning REF CURSORs to JDBC applications
- Error support:
  - RAISE_APPLICATION_ERROR procedure
  - RAISE statement
  - SQLCODE function
  - SQLERRM function

## Creating PL/SQL procedures and functions from a CLP script

You can create PL/SQL procedures and functions from a DB2 command line processor (CLP) script.

1. Formulate PL/SQL procedure or function definitions within a CLP script file. Terminate each statement with a new line and a forward slash character (/). Other statement termination characters are also supported.
2. Save the file. In this example, the file name is `script.db2`.
3. Execute the script from the CLP. If a forward slash character or a semicolon was used to terminate statements, issue the following command:

   ```
   db2 -td/ -vf script.db2
   ```

If another statement termination character (for example, the @ character) was used in the script file, you must specify that character in the command string. For example:

```
db2 -td@ -vf script.db2
```

The CLP script should execute successfully if there are no syntax errors.

The following example of a CLP script creates a PL/SQL function and procedure, and then calls the PL/SQL procedure.

```
CONNECT TO mydb
/

CREATE TABLE emp (
     name           VARCHAR2(10),
     salary         NUMBER,
     comm           NUMBER,
     tot_comp       NUMBER
)
/

INSERT INTO emp VALUES ('Larry', 1000, 50, 0)
/
INSERT INTO emp VALUES ('Curly', 200, 5, 0)
/
INSERT INTO emp VALUES ('Moe', 10000, 1000, 0)
/

CREATE OR REPLACE FUNCTION emp_comp (
     p_sal          NUMBER,
     p_comm         NUMBER )
RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp
/

CREATE OR REPLACE PROCEDURE update_comp(p_name IN VARCHAR) AS
BEGIN
    UPDATE emp SET tot_comp = emp_comp(salary, comm)
      WHERE name = p_name;
END update_comp
/

CALL update_comp('Curly')
/

SELECT * FROM emp
/

CONNECT RESET
/
```

This script produces the following sample output:

```
CALL update_comp('Curly')

  Return Status = 0

SELECT * FROM emp

NAME       SALARY             COMM               TOT_COMP
---------- ------...--------- ----...----------- --------...-------
Larry                    1000                 50                  0
```

```
Curly                   200               5              4920
Moe                   10000            1000                 0

  3 record(s) selected.
```

Test your new procedures or functions by invoking them. For procedures, use the CALL statement. For functions, execute queries or other SQL statements that contain references to those functions.

## Restrictions on PL/SQL support

It is important to note the restrictions on PL/SQL compilation support before performing PL/SQL compilation, or when troubleshooting PL/SQL compilation or runtime problems.

In this version:

- PL/SQL statement compilation and execution for the following product editions is not supported:
  - DB2 Express-C
- PL/SQL functions and triggers cannot be created in a partitioned database environment.
- The NCLOB data type is not supported for use in PL/SQL statements or in PL/SQL contexts when the database is not defined as a Unicode database. In Unicode databases, the NCLOB data type is mapped to a DB2 DBCLOB data type.
- The XMLTYPE data type is not supported.
- TYPE declaration is not supported in a function, procedure, trigger, or anonymous block.
- The FOR EACH STATEMENT option is not supported for PL/SQL triggers.

## PL/SQL sample schema

Most of the PL/SQL examples are based on a PL/SQL sample schema that represents employees in an organization.

The following script (`plsql_sample.sql`) defines that PL/SQL sample schema.

```
--
--  Script that creates the 'sample' tables, views, procedures,
--  functions, triggers, and so on.
--
--  Create and populate tables used in the documentation examples.
--
--  Create the 'dept' table
--
CREATE TABLE dept (
    deptno          NUMBER(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
    dname           VARCHAR2(14) NOT NULL CONSTRAINT dept_dname_uq UNIQUE,
    loc             VARCHAR2(13)
);
--
--  Create the 'emp' table
--
CREATE TABLE emp (
    empno           NUMBER(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename           VARCHAR2(10),
    job             VARCHAR2(9),
    mgr             NUMBER(4),
    hiredate        DATE,
    sal             NUMBER(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm            NUMBER(7,2),
    deptno          NUMBER(2) CONSTRAINT emp_ref_dept_fk
                        REFERENCES dept(deptno)
);
--
--  Create the 'jobhist' table
--
CREATE TABLE jobhist (
    empno           NUMBER(4) NOT NULL,
    startdate       DATE NOT NULL,
    enddate         DATE,
    job             VARCHAR2(9),
    sal             NUMBER(7,2),
    comm            NUMBER(7,2),
    deptno          NUMBER(2),
    chgdesc         VARCHAR2(80),
    CONSTRAINT jobhist_pk PRIMARY KEY (empno, startdate),
    CONSTRAINT jobhist_ref_emp_fk FOREIGN KEY (empno)
        REFERENCES emp(empno) ON DELETE CASCADE,
    CONSTRAINT jobhist_ref_dept_fk FOREIGN KEY (deptno)
        REFERENCES dept (deptno) ON DELETE SET NULL,
    CONSTRAINT jobhist_date_chk CHECK (startdate <= enddate)
);
--
--  Create the 'salesemp' view
--
CREATE OR REPLACE VIEW salesemp AS
    SELECT empno, ename, hiredate, sal, comm FROM emp WHERE job = 'SALESMAN';
--
--  Sequence to generate values for function 'new_empno'
--
CREATE SEQUENCE next_empno START WITH 8000 INCREMENT BY 1;
--
--  Issue PUBLIC grants
--
GRANT ALL ON emp TO PUBLIC;
GRANT ALL ON dept TO PUBLIC;
GRANT ALL ON jobhist TO PUBLIC;
GRANT ALL ON salesemp TO PUBLIC;
--
--  Load the 'dept' table
--
INSERT INTO dept VALUES (10,'ACCOUNTING','NEW YORK');
INSERT INTO dept VALUES (20,'RESEARCH','DALLAS');
INSERT INTO dept VALUES (30,'SALES','CHICAGO');
INSERT INTO dept VALUES (40,'OPERATIONS','BOSTON');
--
--  Load the 'emp' table
--
INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'17-DEC-80',800,NULL,20);
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'20-FEB-81',1600,300,30);
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'22-FEB-81',1250,500,30);
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'02-APR-81',2975,NULL,20);
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'28-SEP-81',1250,1400,30);
INSERT INTO emp VALUES (7698,'BLAKE','MANAGER',7839,'01-MAY-81',2850,NULL,30);
INSERT INTO emp VALUES (7782,'CLARK','MANAGER',7839,'09-JUN-81',2450,NULL,10);
INSERT INTO emp VALUES (7788,'SCOTT','ANALYST',7566,'19-APR-87',3000,NULL,20);
INSERT INTO emp VALUES (7839,'KING','PRESIDENT',NULL,'17-NOV-81',5000,NULL,10);
INSERT INTO emp VALUES (7844,'TURNER','SALESMAN',7698,'08-SEP-81',1500,0,30);
INSERT INTO emp VALUES (7876,'ADAMS','CLERK',7788,'23-MAY-87',1100,NULL,20);
INSERT INTO emp VALUES (7900,'JAMES','CLERK',7698,'03-DEC-81',950,NULL,30);
INSERT INTO emp VALUES (7902,'FORD','ANALYST',7566,'03-DEC-81',3000,NULL,20);
INSERT INTO emp VALUES (7934,'MILLER','CLERK',7782,'23-JAN-82',1300,NULL,10);
--
--  Load the 'jobhist' table
--
```

```
INSERT INTO jobhist VALUES (7369,'17-DEC-80',NULL,'CLERK',800,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7499,'20-FEB-81',NULL,'SALESMAN',1600,300,30,
  'New Hire');
INSERT INTO jobhist VALUES (7521,'22-FEB-81',NULL,'SALESMAN',1250,500,30,
  'New Hire');
INSERT INTO jobhist VALUES (7566,'02-APR-81',NULL,'MANAGER',2975,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7654,'28-SEP-81',NULL,'SALESMAN',1250,1400,30,
  'New Hire');
INSERT INTO jobhist VALUES (7698,'01-MAY-81',NULL,'MANAGER',2850,NULL,30,
  'New Hire');
INSERT INTO jobhist VALUES (7782,'09-JUN-81',NULL,'MANAGER',2450,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7788,'19-APR-87','12-APR-88','CLERK',1000,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7788,'13-APR-88','04-MAY-89','CLERK',1040,NULL,20,
  'Raise');
INSERT INTO jobhist VALUES (7788,'05-MAY-90',NULL,'ANALYST',3000,NULL,20,
  'Promoted to Analyst');
INSERT INTO jobhist VALUES (7839,'17-NOV-81',NULL,'PRESIDENT',5000,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7844,'08-SEP-81',NULL,'SALESMAN',1500,0,30,
  'New Hire');
INSERT INTO jobhist VALUES (7876,'23-MAY-87',NULL,'CLERK',1100,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7900,'03-DEC-81','14-JAN-83','CLERK',950,NULL,10,
  'New Hire');
INSERT INTO jobhist VALUES (7900,'15-JAN-83',NULL,'CLERK',950,NULL,30,
  'Changed to Dept 30');
INSERT INTO jobhist VALUES (7902,'03-DEC-81',NULL,'ANALYST',3000,NULL,20,
  'New Hire');
INSERT INTO jobhist VALUES (7934,'23-JAN-82',NULL,'CLERK',1300,NULL,10,
  'New Hire');

SET SQLCOMPAT PLSQL;
--
--  Procedure that lists all employees' numbers and names
--  from the 'emp' table using a cursor
--
CREATE OR REPLACE PROCEDURE list_emp
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur IS
        SELECT empno, ename FROM emp ORDER BY empno;
BEGIN
    OPEN emp_cur;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cur INTO v_empno, v_ename;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_cur;
END;
/
--
--  Procedure that selects an employee row given the employee
--  number and displays certain columns
--
CREATE OR REPLACE PROCEDURE select_emp (
    p_empno         IN  NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_dname         dept.dname%TYPE;
    v_disp_date     VARCHAR2(10);
BEGIN
    SELECT ename, hiredate, sal, NVL(comm, 0), dname
        INTO v_ename, v_hiredate, v_sal, v_comm, v_dname
        FROM emp e, dept d
        WHERE empno = p_empno
          AND e.deptno = d.deptno;
    v_disp_date := TO_CHAR(v_hiredate, 'YYYY/MM/DD');
    DBMS_OUTPUT.PUT_LINE('Number    : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name      : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
    DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
    DBMS_OUTPUT.PUT_LINE('Department: ' || v_dname);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
        DBMS_OUTPUT.PUT_LINE(SQLERRM);
        DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
        DBMS_OUTPUT.PUT_LINE(SQLCODE);
END;
/
--
--  Procedure that queries the 'emp' table based on
--  department number and employee number or name. Returns
--  employee number and name as IN OUT parameters and job,
--  hire date, and salary as OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query (
    p_deptno        IN     NUMBER,
    p_empno         IN OUT NUMBER,
    p_ename         IN OUT VARCHAR2,
    p_job           OUT    VARCHAR2,
    p_hiredate      OUT    DATE,
    p_sal           OUT    NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
          AND (empno = p_empno
           OR  ename = UPPER(p_ename));
END;
/
--
--  Procedure to call 'emp_query_caller' with IN and IN OUT
--  parameters. Displays the results received from IN OUT and
--  OUT parameters.
--
CREATE OR REPLACE PROCEDURE emp_query_caller
IS
    v_deptno        NUMBER(2);
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER;
BEGIN
    v_deptno := 30;
    v_empno  := 0;
    v_ename  := 'Martin';
    emp_query(v_deptno, v_empno, v_ename, v_job, v_hiredate, v_sal);
    DBMS_OUTPUT.PUT_LINE('Department : ' || v_deptno);
    DBMS_OUTPUT.PUT_LINE('Employee No: ' || v_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('More than one employee was selected');
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No employees were selected');
END;
/
--
--  Function to compute yearly compensation based on semimonthly
--  salary
--
CREATE OR REPLACE FUNCTION emp_comp (
    p_sal           NUMBER,
    p_comm          NUMBER
) RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END;
/
--
--  After statement-level triggers that display a message after
--  an insert, update, or deletion to the 'emp' table. One message
--  per SQL command is displayed.
--
CREATE OR REPLACE TRIGGER user_ins_audit_trig
    AFTER INSERT ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' added employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
CREATE OR REPLACE TRIGGER user_upd_audit_trig
    AFTER UPDATE ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' updated employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
CREATE OR REPLACE TRIGGER user_del_audit_trig
    AFTER DELETE ON emp
    FOR EACH ROW
DECLARE
    v_action        VARCHAR2(24);
BEGIN
    v_action := ' deleted employee(s) on ';
    DBMS_OUTPUT.PUT_LINE('User ' || USER || v_action ||
      TO_CHAR(SYSDATE,'YYYY-MM-DD'));
END;
/
```

```
--
--  Before row-level triggers that display employee number and
--  salary of an employee that is about to be added, updated,
--  or deleted in the 'emp' table
--
CREATE OR REPLACE TRIGGER emp_ins_sal_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Inserting employee ' || :NEW.empno);
    DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
END;
/
CREATE OR REPLACE TRIGGER emp_upd_sal_trig
    BEFORE UPDATE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    sal_diff := :NEW.sal - :OLD.sal;
    DBMS_OUTPUT.PUT_LINE('Updating employee ' || :OLD.empno);
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
    DBMS_OUTPUT.PUT_LINE('..New salary: ' || :NEW.sal);
    DBMS_OUTPUT.PUT_LINE('..Raise     : ' || sal_diff);
END;
/
CREATE OR REPLACE TRIGGER emp_del_sal_trig
    BEFORE DELETE ON emp
    FOR EACH ROW
DECLARE
    sal_diff       NUMBER;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Deleting employee ' || :OLD.empno);
    DBMS_OUTPUT.PUT_LINE('..Old salary: ' || :OLD.sal);
END;
/
--
--  Package specification for the 'emp_admin' package
--
CREATE OR REPLACE PACKAGE emp_admin
IS
    FUNCTION get_dept_name (
        p_deptno        NUMBER
    ) RETURN VARCHAR2;
    FUNCTION update_emp_sal (
        p_empno         NUMBER,
        p_raise         NUMBER
    ) RETURN NUMBER;
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    );
    PROCEDURE fire_emp (
        p_empno         NUMBER
    );
END emp_admin;
/
--
--  Package body for the 'emp_admin' package
```

```
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
    --
    --  Function that queries the 'dept' table based on the department
    --  number and returns the corresponding department name
    --
    FUNCTION get_dept_name (
        p_deptno        IN NUMBER
    ) RETURN VARCHAR2
    IS
        v_dname         VARCHAR2(14);
    BEGIN
        SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
        RETURN v_dname;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
            RETURN '';
    END;
    --
    --  Function that updates an employee's salary based on the
    --  employee number and salary increment/decrement passed
    --  as IN parameters. Upon successful completion the function
    --  returns the new updated salary.
    --
    FUNCTION update_emp_sal (
        p_empno         IN NUMBER,
        p_raise         IN NUMBER
    ) RETURN NUMBER
    IS
        v_sal           NUMBER := 0;
    BEGIN
        SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
        v_sal := v_sal + p_raise;
        UPDATE emp SET sal = v_sal WHERE empno = p_empno;
        RETURN v_sal;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
            RETURN -1;
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
            DBMS_OUTPUT.PUT_LINE(SQLERRM);
            DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
            DBMS_OUTPUT.PUT_LINE(SQLCODE);
            RETURN -1;
    END;
    --
    --  Procedure that inserts a new employee record into the 'emp' table
    --
    PROCEDURE hire_emp (
        p_empno         NUMBER,
        p_ename         VARCHAR2,
        p_job           VARCHAR2,
        p_sal           NUMBER,
        p_hiredate      DATE,
        p_comm          NUMBER,
        p_mgr           NUMBER,
        p_deptno        NUMBER
    )
    AS
    BEGIN
        INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
            VALUES(p_empno, p_ename, p_job, p_sal,
                   p_hiredate, p_comm, p_mgr, p_deptno);
    END;
    --
    --  Procedure that deletes an employee record from the 'emp' table based
    --  on the employee number
    --
    PROCEDURE fire_emp (
        p_empno         NUMBER
    )
    AS
    BEGIN
        DELETE FROM emp WHERE empno = p_empno;
    END;
END;
/

SET SQLCOMPAT DB2;
```

## Obfuscation

Obfuscation encodes the body of the DDL statements for database objects such as routines, triggers, views, and PL/SQL packages. Obfuscating your code helps protect your intellectual property because users cannot read the code, but DB2 Database for Linux®, UNIX®, and Windows® can still understand it.

The DBMS_DDL module provides two routines for obfuscating your routines, triggers, views, or your PL/SQL packages:

**WRAP function**

Takes a routine, trigger, PL/SQL package, or PL/SQL package body definition as an argument and produces a string containing the initial header followed by an obfuscated version of the rest of the statement. For example, input like:

```
CREATE PROCEDURE P(a INT)
BEGIN
  INSERT INTO T1 VALUES (a);
END
```

using the DBMS_DDL.WRAP function might result in:

```
CREATE PROCEDURE P(a INT) WRAPPED SQL09072
aBcDefg12AbcasHGJG6JKHhgkjFGHHkkkljljk878979HJHui99
```

The obfuscated portion of the DDL statement contains codepage invariant characters, ensuring that it is valid for any codepage.

**CREATE_WRAPPED procedure**

Takes the same input as the WRAP function described above, but instead of returning the obfuscated text, an object is created in the database. Internally the object is not obfuscated so that it can be processed by the compiler, but in catalog views like SYSCAT.ROUTINES or SYSCAT.TRIGGERS the content of the TEXT column is obfuscated.

An obfuscated statement can be used in CLP scripts and can be submitted as dynamic SQL using other client interfaces.

Obfuscation is available for the following statements:

- db2look by using the -wrap option
- CREATE FUNCTION
- CREATE PACKAGE
- CREATE PACKAGE BODY

- CREATE PROCEDURE
- CREATE TRIGGER
- CREATE VIEW
- ALTER MODULE

The db2look tool obfuscates all the above statements when the -wrap option is used.

# Blocks (PL/SQL)

PL/SQL block structures can be included within PL/SQL procedure, function, or trigger definitions or executed independently as an anonymous block statement.

PL/SQL block structures and the anonymous block statement contain one or more of the following sections:

- An optional declaration section
- A mandatory executable section
- An optional exception section

These sections can include SQL statements, PL/SQL statements, data type and variable declarations, or other PL/SQL language elements.

## Anonymous block statement (PL/SQL)

The PL/SQL anonymous block statement is an executable statement that can contain PL/SQL control statements and SQL statements. It can be used to implement procedural logic in a scripting language. In PL/SQL contexts, this statement can be compiled and executed by the DB2 data server.

The anonymous block statement, which does not persist in the database, can consist of up to three sections: an optional declaration section, a mandatory executable section, and an optional exception section.

The optional declaration section, which can contain the declaration of variables, cursors, and types that are to be used by statements within the executable and exception sections, is inserted before the executable BEGIN-END block.

The optional exception section can be inserted near the end of the BEGIN-END block. The exception section must begin with the keyword EXCEPTION, and continues until the end of the block in which it appears.

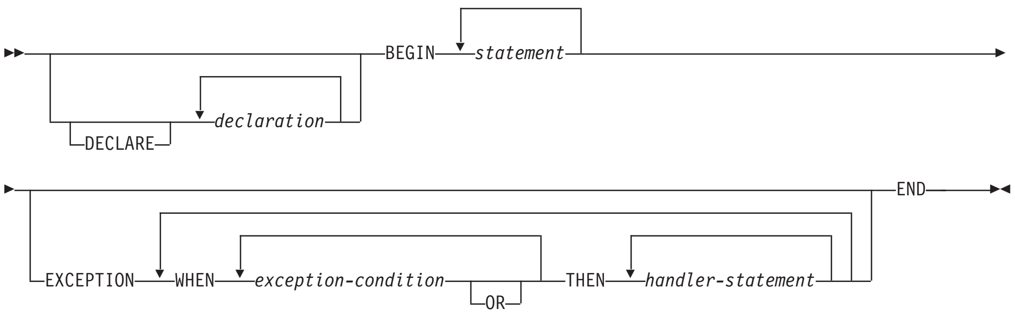
### Invocation

This statement can be executed from an interactive tool or command line interface such as the CLP. This statement can also be embedded within a PL/SQL procedure definition, function definition, or trigger definition. Within these contexts, the statement is called a block structure instead of an anonymous block statement.

### Authorization

No privileges are required to invoke an anonymous block. However, the privileges held by the authorization ID of the statement must include all necessary privileges to invoke the SQL statements that are embedded within the anonymous block.

## Syntax



## Description

**DECLARE**
An optional keyword that starts the DECLARE statement, which can be used to declare data types, variables, or cursors. The use of this keyword depends upon the context in which the block appears.

*declaration*
Specifies a variable, cursor, or type declaration whose scope is local to the block. Each declaration must be terminated by a semicolon.

**BEGIN**
A mandatory keyword that introduces the executable section, which can include one or more SQL or PL/SQL statements. A BEGIN-END block can contain nested BEGIN-END blocks.

*statement*
Specifies a PL/SQL or SQL statement. Each statement must be terminated by a semicolon.

**EXCEPTION**
An optional keyword that introduces the exception section.

**WHEN** *exception-condition*
Specifies a conditional expression that tests for one or more types of exceptions.

**THEN** *handler-statement*
Specifies a PL/SQL or SQL statement that is executed if a thrown exception matches an exception in *exception-condition*. Each statement must be terminated by a semicolon.

**END**
A mandatory keyword that ends the block.

## Examples

The following example shows the simplest possible anonymous block statement that the DB2 data server can compile:

```
BEGIN
    NULL;
END;
```

The following example shows an anonymous block that you can enter interactively through the DB2 CLP:

```
SET SERVEROUTPUT ON;

BEGIN
  dbms_output.put_line( 'Hello' );
END;
```

The following example shows an anonymous block with a declaration section that you can enter interactively through the DB2 CLP:

```
SET SERVEROUTPUT ON;

DECLARE
   current_date DATE := SYSDATE;
BEGIN
     dbms_output.put_line( current_date );
END;
```

# Procedures (PL/SQL)

The DB2 data server supports the compilation and execution of PL/SQL procedures. PL/SQL procedures are database objects that contain PL/SQL procedural logic and SQL statements that can be invoked in contexts where the CALL statement or procedure references are valid.

PL/SQL procedures are created by executing the PL/SQL CREATE PROCEDURE statement. Such procedures can be dropped from the database by using the DB2 SQL DROP statement. If you want to replace the implementation for a procedure, you do not need to drop it. You can use the CREATE PROCEDURE statement and specify the OR REPLACE option to replace the procedure implementation.

## CREATE PROCEDURE statement (PL/SQL)

The CREATE PROCEDURE statement defines a procedure that is stored in the database.

### Invocation

This statement can be executed from the DB2 command line processor (CLP), any supported interactive SQL interface, an application, or a routine.

### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- If the schema name of the procedure does not exist, IMPLICIT_SCHEMA authority on the database
- If the schema name of the procedure refers to an existing schema, CREATEIN privilege on the schema
- DBADM authority

The privileges held by the authorization ID of the statement must also include all of the privileges necessary to invoke the SQL statements that are specified in the procedure body.

The authorization ID of the statement must be the owner of the matched procedure if OR REPLACE is specified (SQLSTATE 42501).

## Syntax

```
>>--CREATE--+------------+--PROCEDURE------------------------------------------->
            '-OR REPLACE-'

>--+------------------------------------------------------------------+-------->
   '-(--+--------------------------------------------------------+--)-'
        |  .-,--------------------------------------------------. |
        |  V                .-IN-----.                            | |
        '---parameter-name--+--------+--data-type--+--------------------+-+-'
                            +-OUT----+             '-| default-clause |-'
                            '-IN OUT-'

                      .-------------.
                      V             |                .-----------.
>--+----------------+--+-IS-+----+-------------+-+--BEGIN----statement-+-------------->
   '-READS SQL DATA-'  '-AS-'    '-declaration-'

>--+----------------------------------------------------------------------------+--->
   |             .------------------------------------------------------------. |
   |             |                  .-----------------.                        | |
   |             V                  V                 |          .-----------. | |
   '-EXCEPTION------WHEN--exception----+---------------+--THEN----statement-+-+-'
                                       '-OR--exception-'

>--END--+----------------+-------------------------------------------------------><
        '-procedure-name-'
```

## Description

**PROCEDURE** *procedure-name*

Specifies an identifier for the procedure. The unqualified form of *procedure-name* is an SQL identifier with a maximum length of 128. In dynamic SQL statements, the value of the CURRENT SCHEMA special register is used to qualify an unqualified object name. In static SQL statements, the QUALIFIER precompile or bind option implicitly specifies the qualifier for unqualified object names. The qualified form of *procedure-name* is a schema name followed by a period character and an SQL identifier. If a two-part name is specified, the schema name cannot begin with 'SYS'; otherwise, an error is returned (SQLSTATE 42939).

The name (including an implicit or explicit qualifier), together with the number of parameters, must not identify a procedure that is described in the catalog (SQLSTATE 42723). The unqualified name, together with the number of parameters, is unique within its schema, but does not need to be unique across schemas.

*parameter-name*

Specifies the name of a parameter. The parameter name must be unique for this procedure (SQLSTATE 42734).

*data-type*

Specifies one of the supported PL/SQL data types.

**READS SQL DATA**

Indicates that SQL statements that do not modify SQL data can be included in the procedure. This clause is a DB2 extension.

**IS or AS**
Introduces the procedure body definition.

*declaration*
Specifies one or more variable, cursor, or REF CURSOR type declarations.

**BEGIN**
Introduces the executable block. The BEGIN-END block can contain an EXCEPTION section.

*statement*
Specifies a PL/SQL or SQL statement. The statement must be terminated by a semicolon.

**EXCEPTION**
An optional keyword that introduces the exception section.

**WHEN** *exception-condition*
Specifies a conditional expression that tests for one or more types of exceptions.

*statement*
Specifies a PL/SQL or SQL statement. The statement must be terminated by a semicolon.

**END**
A mandatory keyword that ends the block. You can optionally specify the name of the procedure.

## Notes

The CREATE PROCEDURE statement can be submitted in obfuscated form. In an obfuscated statement, only the procedure name is readable. The rest of the statement is encoded in such a way that it is not readable, but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

## Examples

The following example shows a simple procedure that takes no parameters:

```
CREATE OR REPLACE PROCEDURE simple_procedure
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('That''s all folks!');
END simple_procedure;
```

The following example shows a procedure that takes an IN and an OUT parameter, and that has GOTO statements whose labels are of the standard PL/SQL form (<<**label**>>):

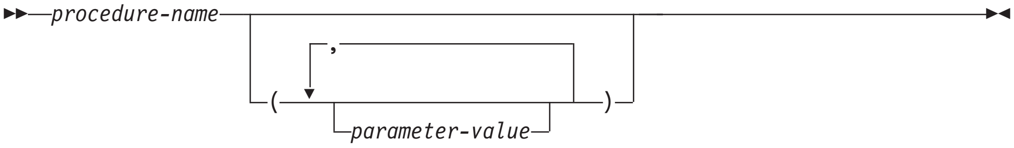
```
CREATE OR REPLACE PROCEDURE test_goto
( p1 IN INTEGER, out1 OUT VARCHAR2(30) )
IS
BEGIN
 <<LABEL2ABOVE>>
 IF p1 = 1 THEN
  out1 := out1 || 'one';
  GOTO LABEL1BELOW;
 END IF;
 if out1 IS NULL THEN
  out1 := out1 || 'two';
  GOTO LABEL2ABOVE;
 END IF;
 out1 := out1 || 'three';

 <<LABEL1BELOW>>
 out1 := out1 || 'four';

END test_goto;
```

## Procedure references (PL/SQL)

Invocation references to PL/SQL procedures within PL/SQL contexts can be compiled by the DB2 data server.

A valid PL/SQL procedure reference consists of the procedure name followed by its parameters, if any.

### Syntax



### Description

*procedure-name*
Specifies an identifier for the procedure.

*parameter-value*
Specifies a parameter value. If no parameters are to be passed, the procedure can be called either with or without parentheses.

### Example

The following example shows how to call a PL/SQL procedure within a PL/SQL context:

```
BEGIN
    simple_procedure;
END;
```

After a PL/SQL procedure has been created in a DB2 database, it can also be called using the CALL statement, which is supported in DB2 SQL contexts and applications using supported DB2 application programming interfaces.

## Function invocation syntax support (PL/SQL)

A number of procedures support function invocation syntax in a PL/SQL assignment statement.

These procedures include:

- DBMS_SQL.EXECUTE
- DBMS_SQL.EXECUTE_AND_FETCH
- DBMS_SQL.FETCH_ROWS
- DBMS_SQL.IS_OPEN
- DBMS_SQL.LAST_ERROR_POSITION
- DBMS_SQL.LAST_ROW_COUNT

- DBMS_SQL.OPEN_CURSOR
- UTL_SMTP.CLOSE_DATA
- UTL_SMTP.COMMAND
- UTL_SMTP.COMMAND_REPLIES
- UTL_SMTP.DATA
- UTL_SMTP.EHLO
- UTL_SMTP.HELO
- UTL_SMTP.HELP
- UTL_SMTP.MAIL
- UTL_SMTP.NOOP
- UTL_SMTP.OPEN_DATA
- UTL_SMTP.QUIT
- UTL_SMTP.RCPT
- UTL_SMTP.RSET
- UTL_SMTP.VRFY

### Examples

```
DECLARE
  cursor1 NUMBER;
  rowsProcessed NUMBER;
BEGIN
  cursor1 := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQL.PARSE(cursor1, 'INSERT INTO T1 VALUES (10)', DBMS_SQL.NATIVE);
  rowsProcessed := DBMS_SQL.EXECUTE(cursor1);
  DBMS_SQL.CLOSE_CURSOR(cursor1);
END;
/
```

```
DECLARE
  v_connection UTL_SMTP.CONNECTION;
  v_reply UTL_SMTP.REPLY;
BEGIN
  UTL_SMTP.OPEN_CONNECTION('127.0.0.1', 25, v_connection, 10, v_reply);
  UTL_SMTP.HELO(v_connection,'127.0.0.1');
  UTL_SMTP.MAIL(v_connection, 'sender1@ca.ibm.com');
  UTL_SMTP.RCPT(v_connection, 'receiver1@ca.ibm.com');
  v_reply := UTL_SMTP.OPEN_DATA (v_connection);
  UTL_SMTP.WRITE_DATA (v_connection, 'Test message');
  UTL_SMTP.CLOSE_DATA (v_connection);
  UTL_SMTP.QUIT(v_connection);
END;
/
```

## Functions (PL/SQL)

The DB2 data server supports the compilation and execution of PL/SQL functions. PL/SQL functions are database objects that contain PL/SQL procedural logic and SQL statements that can be invoked in contexts where expressions are valid. When evaluated, a PL/SQL function returns a value that is substituted within the expression in which the function is embedded.

PL/SQL functions are created by executing the CREATE FUNCTION statement. Such functions can be dropped from the database by using the DB2 SQL DROP statement. If you want to replace the implementation for a function, you do not need to drop it. You can use the CREATE FUNCTION statement and specify the OR REPLACE option to replace the function implementation.

## CREATE FUNCTION statement (PL/SQL)

The CREATE FUNCTION statement defines an SQL scalar function that is stored in the database. A scalar function returns a single value each time it is invoked, and is generally valid wherever an SQL expression is valid. PL/SQL functions do not support output parameters.

### Invocation

This statement can be executed from the DB2 command line processor, any supported interactive SQL interface, an application, or routine.

### Authorization

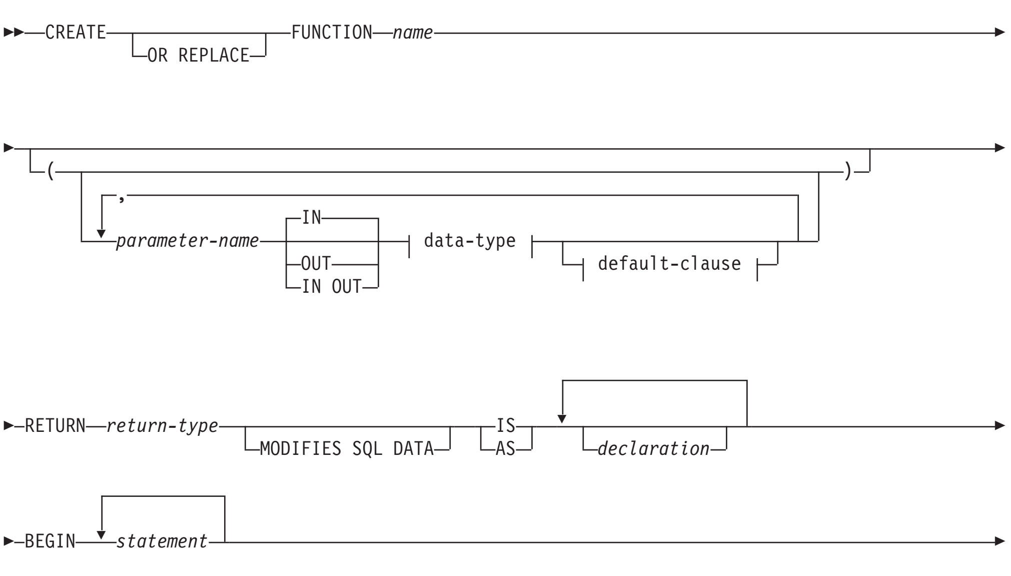
The privileges held by the authorization ID of the statement must include at least one of the following:

- If the schema name of the function does not exist, IMPLICIT_SCHEMA authority on the database
- If the schema name of the function refers to an existing schema, CREATEIN privilege on the schema
- DBADM authority

The privileges held by the authorization ID of the statement must also include all of the privileges necessary to invoke the SQL statements that are specified in the function body.

The authorization ID of the statement must be the owner of the matched function if OR REPLACE is specified (SQLSTATE 42501).

### Syntax



```
►─┬──────────────────────────────────────────────────────────────────────────┬──────────►
  │            ┌─────────────────────────────────────────────────────────┐   │
  │            │                  ┌───────────────────┐          ┌───┐   │   │
  │            ▼                  ▼                   │          ▼   │   │   │
  └─EXCEPTION────WHEN──exception────┬───────────────┬─┴──THEN────statement─┴───┴───┘
                                    └─OR──exception─┘

►─END──┬──────┬──────────────────────────────────────────────────────────────────────►◄
       └─name─┘
```

## Description

The CREATE FUNCTION statement specifies the name of the function, the optional parameters, the return type of the function, and the body of the function. The body of the function is a block that is enclosed by the BEGIN and END keywords. It can contain an optional EXCEPTION section that defines an action to be taken when a defined exception condition occurs.

**OR REPLACE**
Indicates that if a function with the same name already exists in the schema, the new function is to replace the existing one. If this option is not specified, the new function cannot replace an existing one with the same name in the same schema.

**FUNCTION** *name*
Specifies an identifier for the function.

*parameter-name*
Specifies the name of a parameter. The name cannot be the same as any other parameter-name in the parameter list (SQLSTATE 42734).

*data-type*
Specifies one of the supported PL/SQL data types.

**RETURN** *return-type*
Specifies the data type of the scalar value that is returned by the function.

**MODIFIES SQL DATA**
Indicates that the function can issue any SQL statement except statements that are not supported in functions (SQLSTATE 38002 or 42985).

This clause is a DB2 extension. It must be used when dynamic SQL statements that could modify SQL data are specified in statement, otherwise issuing of a dynamic statement that attempts to modify SQL data will fail during function invocation (SQLSTATE 38002).

**IS or AS**
Introduces the block that defines the function body.

*declaration*
Specifies one or more variable, cursor, or REF CURSOR type declarations.

*statement*
Specifies one or more PL/SQL program statements. Each statement must be terminated by a semicolon.

*exception*
Specifies an exception condition name.

### Notes

A PL/SQL function cannot take any action that changes the state of an object that the database manager does not manage.

The CREATE FUNCTION statement can be submitted in obfuscated form. In an obfuscated statement, only the function name is readable. The rest of the statement is encoded in such a way that it is not readable, but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

### Examples

The following example shows a basic function that takes no parameters:

```
CREATE OR REPLACE FUNCTION simple_function
    RETURN VARCHAR2
IS
BEGIN
    RETURN 'That''s All Folks!';
END simple_function;
```

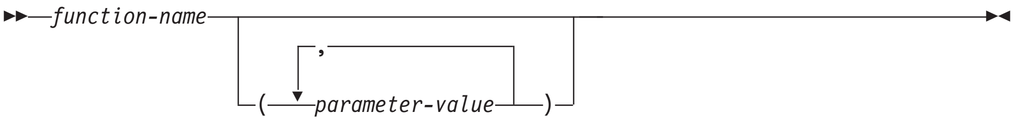
The following example shows a function that takes two input parameters:

```
CREATE OR REPLACE FUNCTION emp_comp (
     p_sal          NUMBER,
     p_comm         NUMBER )
RETURN NUMBER
IS
BEGIN
    RETURN (p_sal + NVL(p_comm, 0)) * 24;
END emp_comp;
```

## Function references (PL/SQL)

PL/SQL functions can be referenced wherever an expression is supported.

### Syntax



### Description

*function-name*
Specifies an identifier for the function.

*parameter-value*
Specifies a value for a parameter.

### Examples

The following example shows how a function named SIMPLE_FUNCTION, defined in the PL/SQL sample schema, can be called from a PL/SQL anonymous block:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(simple_function);
END;
```

The following example shows how a function can be used within an SQL statement:

```
SELECT
   empno "EMPNO", ename "ENAME", sal "SAL", comm "COMM",
   emp_comp(sal, comm) "YEARLY COMPENSATION"
FROM emp
```

# Collections (PL/SQL)

The use of PL/SQL collections is supported by the DB2 data server. A PL/SQL *collection* is a set of ordered data elements with the same data type. Individual data items in the set can be referenced by using subscript notation within parentheses.

In PL/SQL contexts, the DB2 server supports both the VARRAY collection type and associative arrays.

## VARRAY collection type declaration (PL/SQL)

A VARRAY is a type of collection in which each element is referenced by a positive integer called the *array index*. The maximum cardinality of the VARRAY is specified in the type definition.

The TYPE IS VARRAY statement is used to define a VARRAY collection type.

### Syntax

```
►►—TYPE—varraytype—IS VARRAY—(—n—)—OF—datatype—;———————————————————————►◄
```

### Description

*varraytype*
An identifier that is assigned to the array type.

*n* The maximum number of elements in the array type.

*datatype*
A supported data type, such as NUMBER, VARCHAR2, or a record type. The %TYPE attribute and the %ROWTYPE attribute are also supported.

### Example

The following example reads employee names from the EMP table, stores the names in an array variable of type VARRAY, and then displays the results. The EMP table contains one column named ENAME. The code is executed from a DB2 script (`script.db2`). The following commands should be issued from the DB2 command window before executing the script (`db2 -tvf script.db2`):

```
db2set DB2_COMPATIBILITY_VECTOR=FFF
db2stop
db2start
```

The script contains the following code:

```
SET SQLCOMPAT PLSQL;

connect to mydb
/

CREATE PACKAGE foo
AS
```

```
    TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
END;
/

SET SERVEROUTPUT ON
/

DECLARE
    emp_arr         foo.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 5;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..5 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END;
/

DROP PACKAGE foo
/

connect reset
/
```

This script produces the following sample output:

```
Curly
Larry
Moe
Shemp
Joe
```

## CREATE TYPE (VARRAY) statement (PL/SQL)

The CREATE TYPE (VARRAY) statement defines a VARRAY data type.

### Invocation

This statement can be executed from the DB2 command line processor (CLP), any supported interactive SQL interface, an application, or a routine.

### Authorization

The privileges held by the authorization ID of the statement must include at least one of the following:

- If the schema name of the VARRAY type does not exist, IMPLICIT_SCHEMA authority on the database
- If the schema name of the VARRAY type refers to an existing schema, CREATEIN privilege on the schema
- DBADM authority

### Syntax

```
►►─CREATE─┬────────────┬─TYPE─varraytype─┬─IS─┬─VARRAY─(─n─)──────────►◄
          └─OR REPLACE─┘                 └─AS─┘
```

```
►—OF—datatype————————————————————————————————————————►◄
```

### Description

**OR REPLACE**
Indicates that if a user-defined data type with the same name already exists in the schema, the new data type is to replace the existing one. If this option is not specified, the new data type cannot replace an existing one with the same name in the same schema.

*varraytype*
Specifies an identifier for the VARRAY type. The unqualified form of *varraytype* is an SQL identifier with a maximum length of 128. The value of the CURRENT SCHEMA special register is used to qualify an unqualified object name. The qualified form of *varraytype* is a schema name followed by a period character and an SQL identifier. If a two-part name is specified, the schema name cannot begin with 'SYS'; otherwise, an error is returned (SQLSTATE 42939). The name (including an implicit or explicit qualifier) must not identify a user-defined data type that is described in the catalog (SQLSTATE 42723). The unqualified name is unique within its schema, but does not need to be unique across schemas.

*n* Specifies the maximum number of elements in the array type. The maximum cardinality of an array on a given system is limited by the total amount of memory that is available to DB2 applications. As such, although arrays of large cardinalities (up to 2,147,483,647) can be created, not all elements might be available for use.

*datatype*
Specifies a supported data type, such as NUMBER, VARCHAR2, or a record type. The %TYPE attribute and the %ROWTYPE attribute are also supported.

### Example

The following example creates a VARRAY data type with a maximum of 10 elements, where each element has the data type NUMBER:

```
CREATE TYPE NUMARRAY1 AS VARRAY (10) OF NUMBER
```

## Associative arrays (PL/SQL)

A PL/SQL associative array is a collection type that associates a unique key with a value.

An associative array has the following characteristics:

- An associative array type must be defined before array variables of that array type can be declared. Data manipulation occurs in the array variable.
- The array does not need to be initialized; simply assign values to array elements.
- There is no defined limit on the number of elements in the array; it grows dynamically as elements are added.
- The array can be *sparse*; there can be gaps in the assignment of values to keys.
- An attempt to reference an array element that has not been assigned a value results in an exception.

Use the TYPE IS TABLE OF statement to define an associative array type.

## Syntax

```
►►──TYPE──assoctype──IS TABLE OF──┬─datatype─┬──────────────────────────────────►
                                  └─rectype──┘

►─INDEX BY──┬─BINARY_INTEGER──────┬──────────────────────────────────────────────►◄
            ├─PLS_INTEGER─────────┤
            └─VARCHAR2──(──n──)───┘
```

## Description

**TYPE** *assoctype*
    Specifies an identifer for the array type.

*datatype*
    Specifies a scalar data type, such as VARCHAR2 or NUMBER. The %TYPE attribute is also supported.

*rectype*
    Specifies a previously defined record type. The %ROWTYPE attribute is also supported.

**INDEX BY**
    Specifies that the associative array is to be indexed by one of the data types introduced by this clause.

**BINARY INTEGER**
    Integer numeric data.

**PLS_INTEGER**
    Integer numeric data.

**VARCHAR2 (*n*)**
    A variable-length character string of maximum length *n*. The %TYPE attribute is also supported if the object to which the %TYPE attribute is being applied is of the BINARY_INTEGER, PLS_INTEGER, or VARCHAR2 data type.

To declare a variable with an associative array type, specify `array-name assoctype`, where *array-name* represents an identifier that is assigned to the associative array, and *assoctype* represents the identifier for a previously declared array type.

To reference a particular element of the array, specify `array-name(n)`, where *array-name* represents the identifier for a previously declared array, and *n* represents a value of INDEX BY data type of *assoctype*. If the array is defined from a record type, the reference becomes `array-name(n).field`, where *field* is defined within the record type from which the array type is defined. To reference the entire record, omit *field*.

## Examples

The following example reads the first ten employee names from the EMP table, stores them in an array, and then displays the contents of the array.

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type1
IS
    TYPE emp_arr_typ IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
END pkg_test_type1
```

```
/

DECLARE
    emp_arr         pkg_test_type1.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;
END
/
```

This code generates the following sample output:

```
SMITH
ALLEN
WARD
JONES
MARTIN
BLAKE
CLARK
SCOTT
KING
TURNER
```

The example can be modified to use a record type in the array definition.

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type2
IS
    TYPE emp_rec_typ IS RECORD (
        empno       INTEGER,
        ename       VARCHAR2(10)
    );
END pkg_test_type2
/

CREATE OR REPLACE PACKAGE pkg_test_type3
IS
    TYPE emp_arr_typ IS TABLE OF pkg_test_type2.emp_rec_typ INDEX BY BINARY_INTEGER;
END pkg_test_type3
/

DECLARE
    emp_arr         pkg_test_type3.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

The modified code generates the following sample output:

```
EMPNO    ENAME
-----    -------
1001     SMITH
1002     ALLEN
1003     WARD
1004     JONES
1005     MARTIN
1006     BLAKE
1007     CLARK
1008     SCOTT
1009     KING
1010     TURNER
```

This example can be further modified to use the emp%ROWTYPE attribute to define `emp_arr_typ`, instead of using the `emp_rec_typ` record type.

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type4
IS
    TYPE emp_arr_typ IS TABLE OF emp%ROWTYPE INDEX BY BINARY_INTEGER;
END pkg_test_type4
/

DECLARE
    emp_arr         pkg_test_type4.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i).empno := r_emp.empno;
        emp_arr(i).ename := r_emp.ename;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

In this case, the sample output is identical to that of the previous example.

Finally, instead of assigning each field of the record individually, a record-level assignment can be made from `r_emp` to `emp_arr`:

```
SET SERVEROUTPUT ON
/

CREATE OR REPLACE PACKAGE pkg_test_type5
IS
    TYPE emp_rec_typ IS RECORD (
        empno       INTEGER,
        ename       VARCHAR2(10)
    );
END pkg_test_type5
/

CREATE OR REPLACE PACKAGE pkg_test_type6
IS
    TYPE emp_arr_typ IS TABLE OF pkg_test_type5.emp_rec_typ INDEX BY BINARY_INTEGER;
END pkg_test_type6
```

```
/

DECLARE
    emp_arr         pkg_test_type6.emp_arr_typ;
    CURSOR emp_cur IS SELECT empno, ename FROM emp WHERE ROWNUM <= 10;
    i               INTEGER := 0;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp;
    END LOOP;
    FOR j IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j).empno || '     ' ||
            emp_arr(j).ename);
    END LOOP;
END
/
```

## Collection methods (PL/SQL)

Collection methods can be used to obtain information about collections or to modify collections.

The following commands should be executed before attempting to run the examples in Table 4 on page 125.

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
db2 connect to mydb
```

The MYDB database has one table, EMP, which has one column, ENAME (defined as VARCHAR(10)):

```
db2 select * from emp

ENAME
----------
Curly
Larry
Moe
Shemp
Joe

  5 record(s) selected.
```

*Table 4. Collection methods that are supported (or tolerated) by the DB2 data server in a PL/SQL context*

<table>
<tr><th>Collection method</th><th>Description</th><th>Example</th></tr>
<tr><td>COUNT</td><td>Returns the number of elements in a collection.</td><td>

```
CREATE PACKAGE foo
AS
    TYPE sparse_arr_typ IS TABLE OF NUMBER
      INDEX BY BINARY_INTEGER;
END;
/

SET SERVEROUTPUT ON
/

DECLARE
    sparse_arr      foo.sparse_arr_typ;
BEGIN
    sparse_arr(-10)   := -10;
    sparse_arr(0)     := 0;
    sparse_arr(10)    := 10;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
      sparse_arr.COUNT);
END;
/
```

</td></tr>
<tr><td>DELETE</td><td>Removes all elements from a collection.</td><td>

```
CREATE PACKAGE foo
AS
    TYPE names_typ IS TABLE OF VARCHAR2(10)
      INDEX BY BINARY INTEGER;
END;
/

SET SERVEROUTPUT ON
/

DECLARE
    actor_names      foo.names_typ;

BEGIN

    actor_names(1) := 'Chris';
    actor_names(2) := 'Steve';
    actor_names(3) := 'Kate';
    actor_names(4) := 'Naomi';
    actor_names(5) := 'Peter';
    actor_names(6) := 'Philip';
    actor_names(7) := 'Michael';
    actor_names(8) := 'Gary';

    DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
      actor_names.COUNT);

    actor_names.DELETE(2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
      actor_names.COUNT);

    actor_names.DELETE(3, 5);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
      actor_names.COUNT);

    actor_names.DELETE;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' ||
      actor_names.COUNT);

END;
/
```

</td></tr>
</table>

*Table 4. Collection methods that are supported (or tolerated) by the DB2 data server in a PL/SQL context (continued)*

| Collection method | Description | Example |
|---|---|---|
| DELETE (*n*) | Removes element *n* from an associative array. You cannot delete individual elements from a VARRAY collection type. | See "DELETE". |
| DELETE (*n1*, *n2*) | Removes all elements from *n1* to *n2* from an associative array. You cannot delete individual elements from a VARRAY collection type. | See "DELETE". |
| EXISTS (*n*) | Returns TRUE if the specified element exists. | `CREATE PACKAGE foo`<br>`AS`<br>`    TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);`<br>`END;`<br>`/`<br><br>`SET SERVEROUTPUT ON`<br>`/`<br><br>`DECLARE`<br>`    emp_arr         foo.emp_arr_typ;`<br>`    CURSOR emp_cur IS SELECT ename FROM emp`<br>`      WHERE ROWNUM <= 5;`<br>`    i               INTEGER := 0;`<br>`BEGIN`<br>`    FOR r_emp IN emp_cur LOOP`<br>`        i := i + 1;`<br>`        emp_arr(i) := r_emp.ename;`<br>`    END LOOP;`<br>`    emp_arr.TRIM;`<br>`    FOR j IN 1..5 LOOP`<br>`        IF emp_arr.EXISTS(j) = true THEN`<br>`            DBMS_OUTPUT.PUT_LINE(emp_arr(j));`<br>`        ELSE`<br>`            DBMS_OUTPUT.PUT_LINE('THIS ELEMENT`<br>`              HAS BEEN DELETED');`<br>`        END IF;`<br>`    END LOOP;`<br>`END;`<br>`/` |
| EXTEND | Appends a single NULL element to a collection. | No-op |
| EXTEND (*n*) | Appends *n* NULL elements to a collection. | No-op |
| EXTEND (*n1*, *n2*) | Appends *n1* copies of the *n2*th element to a collection. | No-op |

*Table 4. Collection methods that are supported (or tolerated) by the DB2 data server in a PL/SQL context (continued)*

<table>
<tr><th>Collection method</th><th>Description</th><th>Example</th></tr>
<tr><td>FIRST</td><td>Returns the smallest index number in a collection.</td><td><pre>
CREATE PACKAGE foo
AS
    TYPE emp_arr_typ IS VARRAY(5) OF VARCHAR2(10);
END;
/

SET SERVEROUTPUT ON
/

DECLARE
    emp_arr         foo.emp_arr_typ;
    CURSOR emp_cur IS SELECT ename FROM emp
      WHERE ROWNUM <= 5;
    i               INTEGER := 0;
    k               INTEGER := 0;
    l               INTEGER := 0;
BEGIN

    FOR r_emp IN emp_cur LOOP
        i := i + 1;
        emp_arr(i) := r_emp.ename;
    END LOOP;

    -- Use FIRST and LAST to specify the lower and
    --   upper bounds of a loop range:
    FOR j IN emp_arr.FIRST..emp_arr.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(j));
    END LOOP;

    -- Use NEXT(n) to obtain the subscript of
    --   the next element:
    k := emp_arr.FIRST;
    WHILE k IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(k));
        k := emp_arr.NEXT(k);
    END LOOP;

    -- Use PRIOR(n) to obtain the subscript of
    --   the previous element:
    l := emp_arr.LAST;
    WHILE l IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE(emp_arr(l));
        l := emp_arr.PRIOR(l);
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);

    emp_arr.TRIM;
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);

    emp_arr.TRIM(2);
    DBMS_OUTPUT.PUT_LINE('COUNT: ' || emp_arr.COUNT);

    DBMS_OUTPUT.PUT_LINE('Max. no. elements = ' ||
      emp_arr.LIMIT);

END;
/
</pre></td></tr>
<tr><td>LAST</td><td>Returns the largest index number in a collection.</td><td>See “FIRST”.</td></tr>
</table>

*Table 4. Collection methods that are supported (or tolerated) by the DB2 data server in a PL/SQL context (continued)*

| Collection method | Description | Example |
|---|---|---|
| LIMIT | Returns the maximum number of elements for a VARRAY, or NULL for nested tables. | See "FIRST". |
| NEXT (*n*) | Returns the index number of the element immediately following the specified element. | See "FIRST". |
| PRIOR (*n*) | Returns the index number of the element immediately prior to the specified element. | See "FIRST". |
| TRIM | Removes a single element from the end of a collection. You cannot trim elements from an associative array collection type. | See "FIRST". |
| TRIM (*n*) | Removes *n* elements from the end of a collection. You cannot trim elements from an associative array collection type. | See "FIRST". |

# Variables (PL/SQL)

Variables must be declared before they are referenced.

Variables that are used in a block must generally be defined in the declaration section of the block unless they are global variables or package-level variables. The declaration section contains the definitions of variables, cursors, and other types that can be used in PL/SQL statements within the block. A variable declaration consists of a name that is assigned to the variable and the data type of the variable. Optionally, the variable can be initialized to a default value within the variable declaration.

Procedures and functions can have parameters for passing input values. Procedures can also have parameters for passing output values, or parameters for passing both input and output values.

PL/SQL also includes variable data types to match the data types of existing columns, rows, or cursors using the %TYPE and %ROWTYPE qualifiers.

## Variable declarations (PL/SQL)

Variables that are used in a block must generally be defined in the declaration section of the block unless they are global variables or package-level variables. A variable declaration consists of a name that is assigned to the variable and the data type of the variable. Optionally, the variable can be initialized to a default value within the variable declaration.

### Syntax

```
►►──name──┬──────────┬──type──┬──────────┬──┬─────────────────────────────┬──────►◄
          └─CONSTANT─┘        └─NOT NULL─┘  └─┬─:=──────┬──┬─expression─┬─┘
                                              └─DEFAULT─┘  └─NULL───────┘
```

### Description

*name*
Specifies an identifier that is assigned to the variable.

**CONSTANT**
Specifies that the variable value is constant. A default expression must be assigned, and a new value cannot be assigned to the variable within the application program.

*type*
Specifies a data type for the variable.

**NOT NULL**
Specifies that the variable cannot have a null value. If NOT NULL is specified, a default expression must be assigned, and the variable cannot be made null within the application program.

**DEFAULT**
Specifies a default value for the variable. This default is evaluated every time that the block is entered. For example, if SYSDATE has been assigned to a variable of type DATE, the variable resolves to the current invocation time, not to the time at which the procedure or function was precompiled.

**:=** The assignment operator is a synonym for the DEFAULT keyword. However, if this operator is specified without *expression*, the variable is initialized to the value NULL.

*expression*
Specifies the initial value that is to be assigned to the variable when the block is entered.

**NULL**
Specifies the SQL value NULL, which has a null value.

### Example

The following procedure shows variable declarations that utilize defaults consisting of string and numeric expressions:

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno        NUMBER
)
IS
    todays_date     DATE := SYSDATE;
    rpt_title       VARCHAR2(60) := 'Report For Department # ' || p_deptno
             || ' on ' || todays_date;
    base_sal        INTEGER := 35525;
    base_comm_rate  NUMBER := 1.33333;
    base_annual     NUMBER := ROUND(base_sal * base_comm_rate, 2);
BEGIN
    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || base_annual);
END;
```

The following sample output was obtained by calling this procedure:

```
CALL dept_salary_rpt(20);

Report For Department # 20 on 10-JUL-07 16:44:45
Base Annual Salary: 47366.55
```

## Parameter modes (PL/SQL)

PL/SQL procedure parameters can have one of three possible modes: IN, OUT, or IN OUT. PL/SQL function parameters can only be IN.

- An IN formal parameter is initialized to the actual parameter with which it was called, unless it was explicitly initialized with a default value. The IN parameter can be referenced within the called program; however, the called program cannot assign a new value to the IN parameter. After control returns to the calling program, the actual parameter always contains the value to which it was set prior to the call.
- An OUT formal parameter is initialized to the actual parameter with which it was called. The called program can reference and assign new values to the formal parameter. If the called program terminates without an exception, the actual parameter takes on the value to which the formal parameter was last set. If a handled exception occurs, the actual parameter takes on the last value to which the formal parameter was set. If an unhandled exception occurs, the value of the actual parameter remains what it was prior to the call.
- Like an IN parameter, an IN OUT formal parameter is initialized to the actual parameter with which it was called. Like an OUT parameter, an IN OUT formal parameter is modifiable by the called program, and the last value of the formal parameter is passed to the calling program's actual parameter if the called program terminates without an exception. If a handled exception occurs, the actual parameter takes on the last value to which the formal parameter was set. If an unhandled exception occurs, the value of the actual parameter remains what it was prior to the call.

Table 5 summarizes this behavior.

*Table 5. Parameter modes*

| Mode property | IN | IN OUT | OUT |
|---|---|---|---|
| Formal parameter initialized to: | Actual parameter value | Actual parameter value | Actual parameter value |
| Formal parameter modifiable by the called program? | No | Yes | Yes |
| After normal termination of the called program, actual parameter contains: | Original actual parameter value prior to the call | Last value of the formal parameter | Last value of the formal parameter |

*Table 5. Parameter modes (continued)*

| Mode property | IN | IN OUT | OUT |
|---|---|---|---|
| After a handled exception in the called program, actual parameter contains: | Original actual parameter value prior to the call | Last value of the formal parameter | Last value of the formal parameter |
| After an unhandled exception in the called program, actual parameter contains: | Original actual parameter value prior to the call | Original actual parameter value prior to the call | Original actual parameter value prior to the call |

## Data types (PL/SQL)

The DB2 data server supports a wide range of data types that can be used to declare variables in a PL/SQL block.

*Table 6. Supported scalar data types that are available in PL/SQL*

| PL/SQL data type | DB2 SQL data type | Description |
|---|---|---|
| BINARY_INTEGER | INTEGER | Integer numeric data |
| BLOB | BLOB (4096) | Binary data |
| BLOB (*n*) | BLOB (*n*)<br>*n* = 1 to 2 147 483 647 | Binary large object data |
| BOOLEAN | BOOLEAN | Logical Boolean (true or false) |
| CHAR | CHAR (1) | Fixed-length character string data of length 1 |
| CHAR (*n*) | CHAR (*n*)<br>*n* = 1 to 254 | Fixed-length character string data of length *n* |
| CHAR VARYING (*n*) | VARCHAR (*n*) | Variable-length character string data of maximum length *n* |
| CHARACTER | CHARACTER (1) | Fixed-length character string data of length 1 |
| CHARACTER (*n*) | CHARACTER (*n*)<br>*n* = 1 to 254 | Fixed-length character string data of length *n* |
| CHARACTER VARYING (*n*) | VARCHAR (*n*)<br>*n* = 1 to 32 672 | Variable-length character string data of maximum length *n* |
| CLOB | CLOB (1M) | Character large object data |
| CLOB (*n*) | CLOB (*n*)<br>*n* = 1 to 2 147 483 647 | Fixed-length long character string data of length *n* |
| DATE | DATE [1] | Date and time data (expressed to the second) |
| DEC | DEC (9, 2) | Decimal numeric data |
| DEC (*p*) | DEC (*p*)<br>*p* = 1 to 31 | Decimal numeric data of precision *p* |
| DEC (*p*, *s*) | DEC (*p*, *s*)<br>*p* = 1 to 31; *s* = 1 to 31 | Decimal numeric data of precision *p* and scale *s* |

*Table 6. Supported scalar data types that are available in PL/SQL (continued)*

| PL/SQL data type | DB2 SQL data type | Description |
|---|---|---|
| DECIMAL | DECIMAL (9, 2) | Decimal numeric data |
| DECIMAL (*p*) | DECIMAL (*p*)<br>*p* = 1 to 31 | Decimal numeric data of precision *p* |
| DECIMAL (*p*, *s*) | DECIMAL (*p*, *s*)<br>*p* = 1 to 31; *s* = 1 to 31 | Decimal numeric data of precision *p* and scale *s* |
| DOUBLE | DOUBLE | Double precision floating-point number |
| DOUBLE PRECISION | DOUBLE PRECISION | Double precision floating-point number |
| FLOAT | FLOAT | Float numeric data |
| FLOAT (*n*)<br>*n* = 1 to 24 | REAL | Real numeric data |
| FLOAT (*n*)<br>*n* = 25 to 53 | DOUBLE | Double numeric data |
| INT | INT | Signed four-byte integer numeric data |
| INTEGER | INTEGER | Signed four-byte integer numeric data |
| LONG | CLOB (32760) | Character large object data |
| LONG RAW | BLOB (32760) | Binary large object data |
| LONG VARCHAR | CLOB (32760) | Character large object data |
| NATURAL | INTEGER | Signed four-byte integer numeric data |
| NCHAR | GRAPHIC (127) | Fixed-length graphic string data |
| NCHAR (*n*)<br>*n* = 1 to 2000 | GRAPHIC (*n*)<br>*n* = 1 to 127 | Fixed-length graphic string data of length *n* |
| NCLOB [2] | DBCLOB (1M) | Double-byte character large object data |
| NCLOB (*n*) | DBCLOB (2000) | Double-byte long character string data of maximum length *n* |
| NVARCHAR2 | VARGRAPHIC (2048) | Variable-length graphic string data |
| NVARCHAR2 (*n*) | VARGRAPHIC (*n*) | Variable-length graphic string data of maximum length *n* |
| NUMBER | NUMBER [3] | Exact numeric data |
| NUMBER (*p*) | NUMBER (*p*) [3] | Exact numeric data of maximum precision *p* |
| NUMBER (*p*, *s*) | NUMBER (*p*, *s*) [3]<br>*p* = 1 to 31 | Exact numeric data of maximum precision *p* and scale *s* |
| NUMERIC | NUMERIC (9.2) | Exact numeric data |
| NUMERIC (*p*) | NUMERIC (*p*)<br>*p* = 1 to 31 | Exact numeric data of maximum precision *p* |

*Table 6. Supported scalar data types that are available in PL/SQL (continued)*

| **PL/SQL data type** | **DB2 SQL data type** | **Description** |
|---|---|---|
| NUMERIC (*p*, s) | NUMERIC (*p*, *s*)<br>*p* = 1 to 31; *s* = 0 to 31 | Exact numeric data of maximum precision *p* and scale *s* |
| PLS_INTEGER | INTEGER | Integer numeric data |
| RAW | BLOB (32767) | Binary large object data |
| RAW (*n*) | BLOB (*n*)<br>*n* = 1 to 32 767 | Binary large object data |
| SMALLINT | SMALLINT | Signed two-byte integer data |
| TIMESTAMP (0) | TIMESTAMP (0) | Date data with timestamp information |
| TIMESTAMP (*p*) | TIMESTAMP (*p*) | Date and time data with optional fractional seconds and precision *p* |
| VARCHAR | VARCHAR (4096) | Variable-length character string data with a maximum length of 4096 characters |
| VARCHAR (*n*) | VARCHAR (*n*) | Variable-length character string data with a maximum length of *n* characters |
| VARCHAR2 (*n*) | VARCHAR2 (*n*) [4] | Variable-length character string data with a maximum length of *n* characters |

1. When the **DB2_COMPATIBILITY_VECTOR** registry variable is set for the DATE data type, DATE is equivalent to TIMESTAMP (0).
2. For restrictions on the NCLOB data type in certain database environments, see "Restrictions on PL/SQL support".
3. This data type is supported when the **number_compat** database configuration parameter set to ON.
4. This data type is supported when the **varchar2_compat** database configuration parameter set to ON.

In addition to the scalar data types described in Table 6 on page 131, the DB2 data server also supports collection types, record types, and REF CURSOR types.

## %TYPE attribute in variable declarations (PL/SQL)

The %TYPE attribute, used in PL/SQL variable and parameter declarations, is supported by the DB2 data server. Use of this attribute ensures that type compatibility between table columns and PL/SQL variables is maintained.

A qualified column name in dot notation or the name of a previously declared variable must be specified as a prefix to the %TYPE attribute. The data type of this column or variable is assigned to the variable being declared. If the data type of the column or variable changes, there is no need to modify the declaration code.

The %TYPE attribute can also be used with formal parameter declarations.

## Syntax

```
►►─name──┬─┬─table─┬─.─column─┬─%TYPE────────────────────────────────────►◄
         │ └─view──┘          │
         └─variable───────────┘
```

## Description

*name*
: Specifies an identifier for the variable or formal parameter that is being declared.

*table*
: Specifies an identifier for the table whose column is to be referenced.

*view*
: Specifies an identifier for the view whose column is to be referenced.

*column*
: Specifies an identifier for the table or view column that is to be referenced.

*variable*
: Specifies an identifier for a previously declared variable that is to be referenced. The variable does not inherit any other column attributes, such as, for example, the nullability attribute.

## Example

The following example shows a procedure that queries the EMP table using an employee number, displays the employee's data, finds the average salary of all employees in the department to which the employee belongs, and then compares the chosen employee's salary with the department average.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN NUMBER
)
IS
    v_ename         VARCHAR2(10);
    v_job           VARCHAR2(9);
    v_hiredate      DATE;
    v_sal           NUMBER(7,2);
    v_deptno        NUMBER(2);
    v_avgsal        NUMBER(7,2);
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
```

```
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

This procedure could be rewritten without explicitly coding the EMP table data types in the declaration section.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_hiredate      emp.hiredate%TYPE;
    v_sal           emp.sal%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_avgsal        v_sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO v_ename, v_job, v_hiredate, v_sal, v_deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || v_ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || v_job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || v_hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || v_sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || v_deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = v_deptno;
    IF v_sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

The p_empno parameter is an example of a formal parameter that is defined using the %TYPE attribute. The v_avgsal variable is an example of the %TYPE attribute referring to another variable instead of a table column.

The following sample output is generated by a call to the EMP_SAL_QUERY procedure:

```
CALL emp_sal_query(7698);

Employee # : 7698
Name       : BLAKE
Job        : MANAGER
Hire Date  : 01-MAY-81 00:00:00
Salary     : 2850.00
Dept #     : 30
Employee's salary is more than the department average of 1566.67
```

## Record variables based on user-defined record types (PL/SQL)

PL/SQL record variable declarations based on user-defined record type definitions are supported by the DB2 data server in PL/SQL contexts.

A *record type* is a definition of a record that consists of one or more identifiers and their corresponding data types. A record type cannot, by itself, be used to manipulate data. You can declare PL/SQL record variables that are based on existing user-defined record types, and you can create user-defined record types by using the PL/SQL TYPE IS RECORD statement. A record type definition is only supported in the CREATE PACKAGE or CREATE PACKAGE BODY statement.

A *record variable* (or record) is an instance of a record type. A record variable is declared from a record type. The properties of the record, such as its field names and types, are inherited from the record type.

Dot notation is used to reference fields in a record. For example, `record.field`.

### Syntax

```
                                  ┌─,──────────────┐
                                  ▼                │
►►──TYPE──rectype──IS RECORD──(─────field──datatype─┴──)──────────────────────►◄
```

### Description

**TYPE** *rectype* **IS RECORD**
Specifies an identifier for the record type.

*field*
Specifies an identifier for a field of the record type.

*datatype*
Specifies the corresponding data type of the *field*. The %TYPE attribute is supported; the %ROWTYPE attribute is not supported.

### Example

The following example shows a package that references a user-defined record type:

```
CREATE OR REPLACE PACKAGE pkg7a
IS
TYPE t1_typ IS RECORD (
  c1 T1.C1%TYPE,
  c2 VARCHAR(10)
);
END;
```

## %ROWTYPE attribute in record type declarations (PL/SQL)

The %ROWTYPE attribute, used to declare PL/SQL variables of type record with fields that correspond to the columns of a table or view, is supported by the DB2 data server. Each field in a PL/SQL record assumes the data type of the corresponding column in the table.

A *record* is a named, ordered collection of fields. A *field* is similar to a variable; it has an identifier and a data type, but it also belongs to a record, and must be referenced using dot notation, with the record name as a qualifier.

### Syntax

```
►►──record──┬─table─┬──%ROWTYPE────────────────────────────────────────────►◄
            └─view──┘
```

## Description

*record*
Specifies an identifier for the record.

*table*
Specifies an identifier for the table whose column definitions will be used to define the fields in the record.

*view*
Specifies an identifier for the view whose column definitions will be used to define the fields in the record.

**%ROWTYPE**
Specifies that the record field data types are to be derived from the column data types that are associated with the identified table or view. Record fields do not inherit any other column attributes, such as, for example, the nullability attribute.

## Example

The following example shows how to use the %ROWTYPE attribute to create a record (named r_emp) instead of declaring individual variables for the columns in the EMP table.

```
CREATE OR REPLACE PROCEDURE emp_sal_query (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
    v_avgsal        emp.sal%TYPE;
BEGIN
    SELECT ename, job, hiredate, sal, deptno
        INTO r_emp.ename, r_emp.job, r_emp.hiredate, r_emp.sal, r_emp.deptno
        FROM emp WHERE empno = p_empno;
    DBMS_OUTPUT.PUT_LINE('Employee # : ' || p_empno);
    DBMS_OUTPUT.PUT_LINE('Name       : ' || r_emp.ename);
    DBMS_OUTPUT.PUT_LINE('Job        : ' || r_emp.job);
    DBMS_OUTPUT.PUT_LINE('Hire Date  : ' || r_emp.hiredate);
    DBMS_OUTPUT.PUT_LINE('Salary     : ' || r_emp.sal);
    DBMS_OUTPUT.PUT_LINE('Dept #     : ' || r_emp.deptno);

    SELECT AVG(sal) INTO v_avgsal
        FROM emp WHERE deptno = r_emp.deptno;
    IF r_emp.sal > v_avgsal THEN
        DBMS_OUTPUT.PUT_LINE('Employee''s salary is more than the department '
            || 'average of ' || v_avgsal);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee''s salary does not exceed the department '
            || 'average of ' || v_avgsal);
    END IF;
END;
```

# Basic statements (PL/SQL)

The programming statements that can be used in a PL/SQL application include: assignment, DELETE, EXECUTE IMMEDIATE, INSERT, NULL, SELECT INTO, and UPDATE.

# NULL statement (PL/SQL)

The NULL statement is an executable statement that does nothing. The NULL statement can act as a placeholder whenever an executable statement is required, but no SQL operation is wanted; for example, within a branch of the IF-THEN-ELSE statement.

## Syntax

```
►►──NULL──────────────────────────────────────────────────────────────────►◄
```

## Examples

The following example shows the simplest valid PL/SQL program that the DB2 data server can compile:

```
BEGIN
    NULL;
END;
```

The following example shows the NULL statement within an IF...THEN...ELSE statement:

```
CREATE OR REPLACE PROCEDURE divide_it (
    p_numerator     IN  NUMBER,
    p_denominator   IN  NUMBER,
    p_result        OUT NUMBER
)
IS
BEGIN
    IF p_denominator = 0 THEN
        NULL;
    ELSE
        p_result := p_numerator / p_denominator;
    END IF;
END;
```

# Assignment statement (PL/SQL)

The assignment statement sets a previously-declared variable or formal OUT or IN OUT parameter to the value of an expression.

## Syntax

```
►►──variable──:=──expression──────────────────────────────────────────────►◄
```

## Description

*variable*
: Specifies an identifier for a previously-declared variable, OUT formal parameter, or IN OUT formal parameter.

*expression*
: Specifies an expression that evaluates to a single value. The data type of this value must be compatible with the data type of *variable*.

## Example

The following example shows assignment statements in the executable section of a procedure:

```
CREATE OR REPLACE PROCEDURE dept_salary_rpt (
    p_deptno        IN   NUMBER,
    p_base_annual   OUT  NUMBER
)
IS
    todays_date     DATE;
    rpt_title       VARCHAR2(60);
    base_sal        INTEGER;
    base_comm_rate  NUMBER;
BEGIN
    todays_date := SYSDATE;
    rpt_title := 'Report For Department # ' || p_deptno || ' on '
        || todays_date;
    base_sal := 35525;
    base_comm_rate := 1.33333;
    p_base_annual := ROUND(base_sal * base_comm_rate, 2);

    DBMS_OUTPUT.PUT_LINE(rpt_title);
    DBMS_OUTPUT.PUT_LINE('Base Annual Salary: ' || p_base_annual);
END
/
```

# EXECUTE IMMEDIATE statement (PL/SQL)

The EXECUTE IMMEDIATE statement prepares an executable form of an SQL statement from a character string form of the statement and then executes the SQL statement. EXECUTE IMMEDIATE combines the basic functions of the PREPARE and EXECUTE statements.

## Invocation

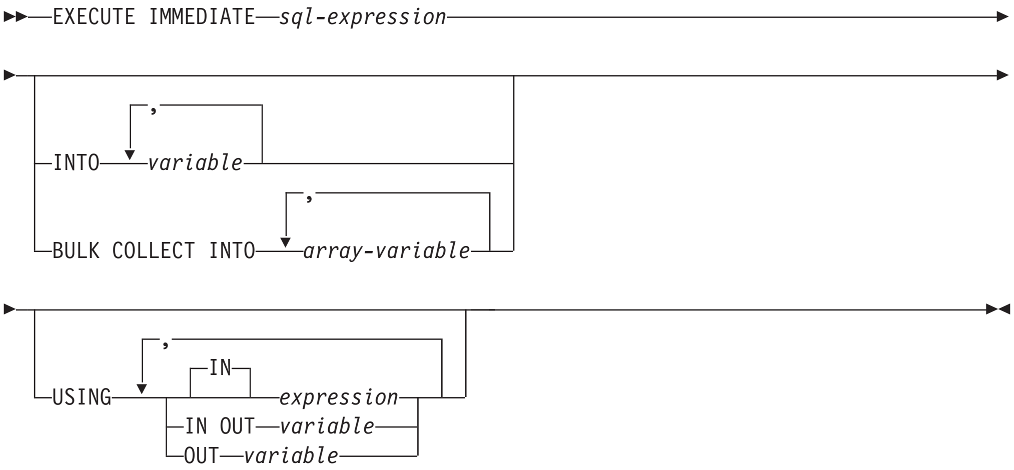
This statement can only be specified in a PL/SQL context.

## Authorization

The authorization rules are those defined for the specified SQL statement.

The authorization ID of the statement might be affected by the DYNAMICRULES bind option.

## Syntax



## Description

*sql-expression*

An expression returning the statement string to be executed. The expression must return a character-string type that is less than the maximum statement size of 2 097 152 bytes. Note that a CLOB(2097152) can contain a maximum size statement, but a VARCHAR cannot.

The statement string must be one of the following SQL statements:

- ALTER
- CALL
- COMMENT
- COMMIT
- Compound SQL (compiled)
- Compound SQL (inlined)
- CREATE
- DECLARE GLOBAL TEMPORARY TABLE
- DELETE
- DROP
- EXPLAIN
- FLUSH EVENT MONITOR
- FLUSH PACKAGE CACHE
- GRANT
- INSERT
- LOCK TABLE
- MERGE
- REFRESH TABLE
- RELEASE SAVEPOINT
- RENAME
- REVOKE
- ROLLBACK
- SAVEPOINT
- SELECT (only when the EXECUTE IMMEDIATE statement also specifies the BULK COLLECT INTO clause)
- SET COMPILATION ENVIRONMENT
- SET CURRENT DECFLOAT ROUNDING MODE
- SET CURRENT DEFAULT TRANSFORM GROUP
- SET CURRENT DEGREE
- SET CURRENT FEDERATED ASYNCHRONY
- SET CURRENT EXPLAIN MODE
- SET CURRENT EXPLAIN SNAPSHOT
- SET CURRENT IMPLICIT XMLPARSE OPTION
- SET CURRENT ISOLATION
- SET CURRENT LOCALE LC_TIME
- SET CURRENT LOCK TIMEOUT
- SET CURRENT MAINTAINED TABLE TYPES FOR OPTIMIZATION
- SET CURRENT MDC ROLLOUT MODE

- SET CURRENT OPTIMIZATION PROFILE
- SET CURRENT QUERY OPTIMIZATION
- SET CURRENT REFRESH AGE
- SET CURRENT SQL_CCFLAGS
- SET ROLE (only if DYNAMICRULES run behavior is in effect for the package)
- SET ENCRYPTION PASSWORD
- SET EVENT MONITOR STATE (only if DYNAMICRULES run behavior is in effect for the package)
- SET INTEGRITY
- SET PASSTHRU
- SET PATH
- SET SCHEMA
- SET SERVER OPTION
- SET SESSION AUTHORIZATION
- SET variable
- TRANSFER OWNERSHIP (only if DYNAMICRULES run behavior is in effect for the package)
- TRUNCATE (only if DYNAMICRULES run behavior is in effect for the package)
- UPDATE

The statement string must not contain a statement terminator, with the exception of compound SQL statements which can contain semicolons (;) to separate statements within the compound block. A compound SQL statement is used within some CREATE and ALTER statements which, therefore, can also contain semicolons.

When an EXECUTE IMMEDIATE statement is executed, the specified statement string is parsed and checked for errors. If the SQL statement is invalid, it is not executed, and an exception is thrown.

**INTO** *variable*
Specifies the name of a variable that is to receive an output value from the corresponding parameter marker.

**BULK COLLECT INTO** *array-variable*
Identifies one or more variables with an array data type. Each row of the query is assigned to an element in each array in the order of the result set, with the array index assigned in sequence.

- If exactly one *array-variable* is specified:
  - If the data type of the *array-variable* element is not a record type, the SELECT list must have exactly one column and the column data type must be assignable to the array element data type.
  - If the data type of the *array-variable* element is a record type, the SELECT list must be assignable to the record type.
- If multiple array variables are specified:
  - The data type of the *array-variable* element must not be a record type.
  - There must be an *array-variable* for each column in the SELECT list.
  - The data type of each column in the SELECT list must be assignable to the array element data type of the corresponding *array-variable*.

If the data type of *array-variable* is an ordinary array, the maximum cardinality must be greater than or equal to the number of rows that are returned by the query.

This clause can only be used if the *sql-expression* is a SELECT statement.

**USING**

**IN** *expression*
Specifies a value that is passed to an input parameter marker. IN is the default.

**IN OUT** *variable*
Specifies the name of a variable that is to provide an input value to, or receive an output value from the corresponding parameter marker.

**OUT** *variable*
Specifies the name of a variable that is to receive an output value from the corresponding parameter marker.

The number and order of evaluated expressions or variables must match the number and order of—and be type-compatible with—the parameter markers in *sql-expression*.

### Notes

- Statement caching affects the behavior of an EXECUTE IMMEDIATE statement.

### Example

```
CREATE OR REPLACE PROCEDURE proc1( p1 IN NUMBER, p2 IN OUT NUMBER, p3 OUT NUMBER )
IS
BEGIN
  p3 := p1 + 1;
  p2 := p2 + 1;
END;
/

EXECUTE IMMEDIATE 'BEGIN proc1( :1, :2, :3 ); END' USING IN p1 + 10, IN OUT p3,
  OUT p2;

EXECUTE IMMEDIATE 'BEGIN proc1( :1, :2, :3 ); END' INTO p3, p2 USING p1 + 10, p3;
```

## SQL statements (PL/SQL)

SQL statements that are supported within PL/SQL contexts can be used to modify data or to specify the manner in which statements are to be executed.

Table 7 lists these statements. The behavior of these statements when executed in PL/SQL contexts is equivalent to the behavior of the corresponding DB2 SQL statements.

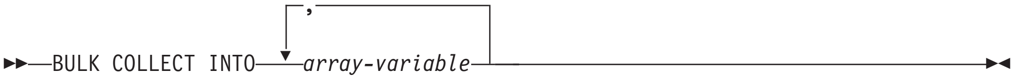
*Table 7. SQL statements that can be executed by the DB2 server within PL/SQL contexts*

| Command | Description |
|---|---|
| DELETE | Deletes rows from a table |
| INSERT | Inserts rows into a table |
| MERGE | Updates a target (a table or view) using data from a source (result of a table reference) |
| SELECT INTO | Retrieves rows from a table |
| UPDATE | Updates rows in a table |

# BULK COLLECT INTO clause (PL/SQL)

A SELECT INTO statement with the optional BULK COLLECT keywords preceding the INTO keyword retrieves multiple rows into an array.

## Syntax



## Description

**BULK COLLECT INTO** *array-variable*

Identifies one or more variables with an array data type. Each row of the result is assigned to an element in each array in the order of the result set, with the array index assigned in sequence.

- If exactly one *array-variable* is specified:
  - If the data type of the *array-variable* element is not a record type, the SELECT list must have exactly one column, and the column data type must be assignable to the array element data type.
  - If the data type of the *array-variable* element is a record type, the SELECT list must be assignable to the record type.
- If multiple array variables are specified:
  - The data type of the *array-variable* element must not be a record type.
  - There must be an *array-variable* for each column in the SELECT list.
  - The data type of each column in the SELECT list must be assignable to the array element data type of the corresponding *array-variable*.

If the data type of *array-variable* is an ordinary array, the maximum cardinality must be greater than or equal to the number of rows that are returned by the query.

## Notes

- Variations of the BULK COLLECT INTO clause are also supported with the FETCH statement and the EXECUTE IMMEDIATE statement.

## Example

The following example shows a procedure that uses the BULK COLLECT INTO clause to return an array of rows from the procedure. The procedure and the type for the array are defined in a package.

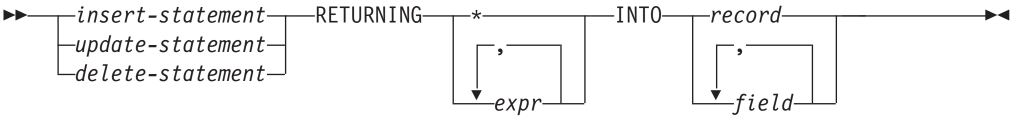
```
CREATE OR REPLACE PACAKGE bci_sample
IS
 TYPE emps_array IS VARRAY (30) OF VARCHAR2(6);

 PROCEDURE get_dept_empno (
  dno       IN   emp.deptno%TYPE,
  emps_dno  OUT  emps_array
  );
END bci_sample;

CREATE OR REPLACE PACKAGE BODY bci_sample
IS
```

```
 PROCEDURE get_dept_empno (
  dno       IN   emp.deptno%TYPE,
  emps_dno  OUT  emps_array
  )
 IS
  BEGIN
   SELECT empno BULK COLLECT INTO emps_dno
    FROM emp
    WHERE deptno=dno;
  END get_dept_empno;
END bci_sample;
```

## RETURNING INTO clause (PL/SQL)

INSERT, UPDATE, and DELETE statements that are appended with the optional RETURNING INTO clause can be compiled by the DB2 data server. When used in PL/SQL contexts, this clause captures the newly added, modified, or deleted values from executing INSERT, UPDATE, or DELETE statements, respectively.

### Syntax



### Description

*insert-statement*
Specifies a valid INSERT statement. An exception is raised if the INSERT statement returns a result set that contains more than one row.

*update-statement*
Specifies a valid UPDATE statement. An exception is raised if the UPDATE statement returns a result set that contains more than one row.

*delete-statement*
Specifies a valid DELETE statement. An exception is raised if the DELETE statement returns a result set that contains more than one row.

**RETURNING ***
Specifies that all of the values from the row that is affected by the INSERT, UPDATE, or DELETE statement are to be made available for assignment.

**RETURNING** *expr*
Specifies an expression to be evaluated against the row that is affected by the INSERT, UPDATE, or DELETE statement. The evaluated results are assigned to a specified record or fields.

**INTO** *record*
Specifies that the returned values are to be stored in a record with compatible fields and data types. The fields must match in number, order, and data type those values that are specified with the RETURNING clause. If the result set contains no rows, the fields in the record are set to the null value.

**INTO** *field*
Specifies that the returned values are to be stored in a set of variables with compatible fields and data types. The fields must match in number, order, and data type those values that are specified with the RETURNING clause. If the result set contains no rows, the fields are set to the null value.

## Examples

The following example shows a procedure that uses the RETURNING INTO clause:

```
CREATE OR REPLACE PROCEDURE emp_comp_update (
    p_empno         IN emp.empno%TYPE,
    p_sal           IN emp.sal%TYPE,
    p_comm          IN emp.comm%TYPE
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_deptno        emp.deptno%TYPE;
BEGIN
    UPDATE emp SET sal = p_sal, comm = p_comm WHERE empno = p_empno
    RETURNING
        empno,
        ename,
        job,
        sal,
        comm,
        deptno
    INTO
        v_empno,
        v_ename,
        v_job,
        v_sal,
        v_comm,
        v_deptno;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Updated Employee # : ' || v_empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || v_ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || v_job);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || v_deptno);
        DBMS_OUTPUT.PUT_LINE('New Salary         : ' || v_sal);
        DBMS_OUTPUT.PUT_LINE('New Commission     : ' || v_comm);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

This procedure returns the following sample output:

```
EXEC emp_comp_update(9503, 6540, 1200);

Updated Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Department         : 40
New Salary         : 6540.00
New Commission     : 1200.00
```

The following example shows a procedure that uses the RETURNING INTO clause with record types:

```
CREATE OR REPLACE PROCEDURE emp_delete (
    p_empno         IN emp.empno%TYPE
)
IS
    r_emp           emp%ROWTYPE;
BEGIN
    DELETE FROM emp WHERE empno = p_empno
    RETURNING
        *
    INTO
        r_emp;

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Deleted Employee # : ' || r_emp.empno);
        DBMS_OUTPUT.PUT_LINE('Name               : ' || r_emp.ename);
        DBMS_OUTPUT.PUT_LINE('Job                : ' || r_emp.job);
        DBMS_OUTPUT.PUT_LINE('Manager            : ' || r_emp.mgr);
        DBMS_OUTPUT.PUT_LINE('Hire Date          : ' || r_emp.hiredate);
        DBMS_OUTPUT.PUT_LINE('Salary             : ' || r_emp.sal);
        DBMS_OUTPUT.PUT_LINE('Commission         : ' || r_emp.comm);
        DBMS_OUTPUT.PUT_LINE('Department         : ' || r_emp.deptno);
    ELSE
        DBMS_OUTPUT.PUT_LINE('Employee # ' || p_empno || ' not found');
    END IF;
END;
```

This procedure returns the following sample output:

```
EXEC emp_delete(9503);

Deleted Employee # : 9503
Name               : PETERSON
Job                : ANALYST
Manager            : 7902
Hire Date          : 31-MAR-05 00:00:00
Salary             : 6540.00
Commission         : 1200.00
Department         : 40
```

## Statement attributes (PL/SQL)

SQL%FOUND, SQL%NOTFOUND, and SQL%ROWCOUNT are PL/SQL attributes that can be used to determine the effect of an SQL statement.

- The SQL%FOUND attribute has a Boolean value that returns TRUE if at least one row was affected by an INSERT, UPDATE, or DELETE statement, or if a SELECT INTO statement retrieved one row. The following example shows an anonymous block in which a row is inserted and a status message is displayed.

  ```
  BEGIN
      INSERT INTO emp (empno,ename,job,sal,deptno)
          VALUES (9001, 'JONES', 'CLERK', 850.00, 40);
      IF SQL%FOUND THEN
          DBMS_OUTPUT.PUT_LINE('Row has been inserted');
      END IF;
  END;
  ```

- The SQL%NOTFOUND attribute has a Boolean value that returns TRUE if no rows were affected by an INSERT, UPDATE, or DELETE statement, or if a SELECT INTO statement did not retrieve a row. For example:

  ```
  BEGIN
      UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9000;
      IF SQL%NOTFOUND THEN
          DBMS_OUTPUT.PUT_LINE('No rows were updated');
      END IF;
  END;
  ```

- The SQL%ROWCOUNT attribute has an integer value that represents the number of rows that were affected by an INSERT, UPDATE, or DELETE statement. For example:

  ```
  BEGIN
      UPDATE emp SET hiredate = '03-JUN-07' WHERE empno = 9001;
      DBMS_OUTPUT.PUT_LINE('# rows updated: ' || SQL%ROWCOUNT);
  END;
  ```

# Control statements (PL/SQL)

Control statements are the programming statements that make PL/SQL a full procedural complement to SQL.

A number of PL/SQL control statements can be compiled by the DB2 data server.

## IF statement (PL/SQL)

Use the IF statement within PL/SQL contexts to execute SQL statements on the basis of certain criteria.

The four forms of the IF statement are:

- IF...THEN...END IF
- IF...THEN...ELSE...END IF
- IF...THEN...ELSE IF...END IF
- IF...THEN...ELSIF...THEN...ELSE...END IF

### IF...THEN...END IF

The syntax of this statement is:

```
IF boolean-expression THEN
  statements
END IF;
```

IF...THEN statements are the simplest form of IF. The statements between THEN and END IF are executed only if the condition evaluates to TRUE. In the following example, an IF...THEN statement is used to test for and to display those employees who have a commission.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

This program generates the following sample output:

```
EMPNO    COMM
-----    -------
7499     $300.00
7521     $500.00
7654    $1400.00
```

## IF...THEN...ELSE...END IF

The syntax of this statement is:

```
IF boolean-expression THEN
  statements
ELSE
  statements
END IF;
```

IF...THEN...ELSE statements specify an alternative set of statements that should be executed if the condition evaluates to FALSE. In the following example, the previous example is modified so that an IF...THEN...ELSE statement is used to display the text “Non-commission” if an employee does not have a commission.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comm          emp.comm%TYPE;
    CURSOR emp_cursor IS SELECT empno, comm FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    COMM');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
            DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
            TO_CHAR(v_comm,'$99999.99'));
        ELSE
            DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || 'Non-commission');
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

This program generates the following sample output:

```
EMPNO    COMM
-----    -------
7369     Non-commission
7499  $   300.00
7521  $   500.00
7566     Non-commission
7654  $  1400.00
7698     Non-commission
7782     Non-commission
7788     Non-commission
7839     Non-commission
7844     Non-commission
7876     Non-commission
7900     Non-commission
7902     Non-commission
7934     Non-commission
```

## IF...THEN...ELSE IF...END IF

The syntax of this statement is:

```
IF boolean-expression THEN
  IF boolean-expression THEN
    statements
```

```
ELSE
  IF boolean-expression THEN
    statements
END IF;
```

You can nest IF statements so that alternative IF statements are invoked, depending on whether the conditions of an outer IF statement evaluate to TRUE or FALSE. In the following example, the outer IF...THEN...ELSE statement tests whether or not an employee has a commission. The inner IF...THEN...ELSE statements subsequently test whether the employee's total compensation exceeds or is less than the company average. When you use this form of the IF statement, you are actually nesting an IF statement inside of the ELSE part of an outer IF statement. You therefore need one END IF for each nested IF and one for the parent IF...ELSE. (Note that the logic in this program can be simplified considerably by calculating each employee's yearly compensation using an NVL function within the SELECT statement of the cursor declaration; however, the purpose of this example is to demonstrate how IF statements can be used.)

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_sal           emp.sal%TYPE;
    v_comm          emp.comm%TYPE;
    v_avg           NUMBER(7,2);
    CURSOR emp_cursor IS SELECT empno, sal, comm FROM emp;
BEGIN
--
--  Calculate the average yearly compensation
--
    SELECT AVG((sal + NVL(comm,0)) * 24) INTO v_avg FROM emp;
    DBMS_OUTPUT.PUT_LINE('Average Yearly Compensation: ' ||
        TO_CHAR(v_avg,'$999,999.99'));
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    YEARLY COMP');
    DBMS_OUTPUT.PUT_LINE('-----    -----------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_sal, v_comm;
        EXIT WHEN emp_cursor%NOTFOUND;
--
--  Test whether or not the employee gets a commission
--
        IF v_comm IS NOT NULL AND v_comm > 0 THEN
--
--  Test whether the employee's compensation with commission exceeds
--  the company average
--
            IF (v_sal + v_comm) * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR((v_sal + v_comm) * 24,'$999,999.99') ||
                    ' Below Average');
            END IF;
        ELSE
--
--  Test whether the employee's compensation without commission exceeds
--  the company average
--
            IF v_sal * 24 > v_avg THEN
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Exceeds Average');
            ELSE
                DBMS_OUTPUT.PUT_LINE(v_empno || '  ' ||
                    TO_CHAR(v_sal * 24,'$999,999.99') || ' Below Average');
            END IF;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
END;
```

This program generates the following sample output:

```
Average Yearly Compensation: $  53,528.57
EMPNO    YEARLY COMP
-----    -----------
7369  $  19,200.00 Below Average
7499  $  45,600.00 Below Average
7521  $  42,000.00 Below Average
7566  $  71,400.00 Exceeds Average
7654  $  63,600.00 Exceeds Average
7698  $  68,400.00 Exceeds Average
7782  $  58,800.00 Exceeds Average
7788  $  72,000.00 Exceeds Average
7839  $ 120,000.00 Exceeds Average
7844  $  36,000.00 Below Average
7876  $  26,400.00 Below Average
7900  $  22,800.00 Below Average
7902  $  72,000.00 Exceeds Average
7934  $  31,200.00 Below Average
```

## IF...THEN...ELSIF...THEN...ELSE...END IF

The syntax of this statement is:

```
IF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements ] ...]
[ ELSE
  statements ]
END IF;
```

IF...THEN...ELSIF...ELSE statements provide the means for checking many alternatives in one statement. Formally, this statement is equivalent to nested IF...THEN...ELSE...IF...THEN statements, but only one END IF is needed. The following example uses an IF...THEN...ELSIF...ELSE statement to count the number of employees by compensation, in steps of $25,000.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_comp          NUMBER(8,2);
    v_lt_25K        SMALLINT := 0;
    v_25K_50K       SMALLINT := 0;
    v_50K_75K       SMALLINT := 0;
    v_75K_100K      SMALLINT := 0;
    v_ge_100K       SMALLINT := 0;
    CURSOR emp_cursor IS SELECT empno, (sal + NVL(comm,0)) * 24 FROM emp;
BEGIN
    OPEN emp_cursor;
    LOOP
        FETCH emp_cursor INTO v_empno, v_comp;
        EXIT WHEN emp_cursor%NOTFOUND;
        IF v_comp < 25000 THEN
            v_lt_25K := v_lt_25K + 1;
        ELSIF v_comp < 50000 THEN
            v_25K_50K := v_25K_50K + 1;
        ELSIF v_comp < 75000 THEN
            v_50K_75K := v_50K_75K + 1;
        ELSIF v_comp < 100000 THEN
```

```
            v_75K_100K := v_75K_100K + 1;
        ELSE
            v_ge_100K := v_ge_100K + 1;
        END IF;
    END LOOP;
    CLOSE emp_cursor;
    DBMS_OUTPUT.PUT_LINE('Number of employees by yearly compensation');
    DBMS_OUTPUT.PUT_LINE('Less than 25,000 : ' || v_lt_25K);
    DBMS_OUTPUT.PUT_LINE('25,000 - 49,9999 : ' || v_25K_50K);
    DBMS_OUTPUT.PUT_LINE('50,000 - 74,9999 : ' || v_50K_75K);
    DBMS_OUTPUT.PUT_LINE('75,000 - 99,9999 : ' || v_75K_100K);
    DBMS_OUTPUT.PUT_LINE('100,000 and over : ' || v_ge_100K);
END;
```

This program generates the following sample output:

```
Number of employees by yearly compensation
Less than 25,000 : 2
25,000 - 49,9999 : 5
50,000 - 74,9999 : 6
75,000 - 99,9999 : 0
100,000 and over : 1
```

# CASE statement (PL/SQL)

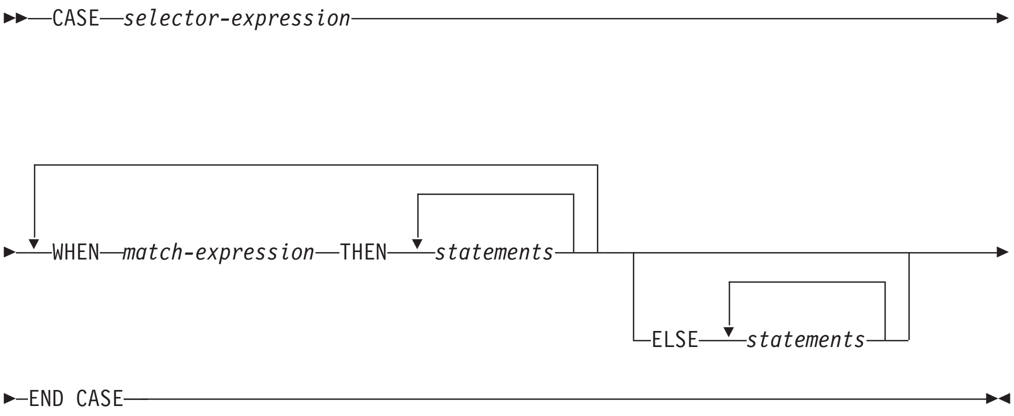
The CASE statement executes a set of one or more statements when a specified search condition is true. CASE is a standalone statement that is distinct from the CASE expression, which must appear as part of an expression.

There are two forms of the CASE statement: the simple CASE statement and the searched CASE statement.

## Simple CASE statement (PL/SQL)

The simple CASE statement attempts to match an expression (known as the *selector*) to another expression that is specified in one or more WHEN clauses. A match results in the execution of one or more corresponding statements.

### Syntax



### Description

**CASE** *selector-expression*
    Specifies an expression whose value has a data type that is compatible with

each *match-expression*. If the value of *selector-expression* matches the first *match-expression*, the statements in the corresponding THEN clause are executed. If there are no matches, the statements in the corresponding ELSE clause are executed. If there are no matches and there is no ELSE clause, an exception is thrown.

**WHEN** *match-expression*
Specifies an expression that is evaluated within the CASE statement. If *selector-expression* matches a *match-expression*, the statements in the corresponding THEN clause are executed.

**THEN**
A keyword that introduces the statements that are to be executed when the corresponding Boolean expression evaluates to TRUE.

*statements*
Specifies one or more SQL or PL/SQL statements, each terminated with a semicolon.

**ELSE**
A keyword that introduces the default case of the CASE statement.

## Example

The following example uses a simple CASE statement to assign a department name and location to a variable that is based upon the department number.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE v_deptno
            WHEN 10 THEN v_dname := 'Accounting';
                         v_loc   := 'New York';
            WHEN 20 THEN v_dname := 'Research';
                         v_loc   := 'Dallas';
            WHEN 30 THEN v_dname := 'Sales';
                         v_loc   := 'Chicago';
            WHEN 40 THEN v_dname := 'Operations';
                         v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                         v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

This program returns the following sample output:

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
7934     MILLER      10      Accounting     New York
```

## Searched CASE statement (PL/SQL)

A searched CASE statement uses one or more Boolean expressions to determine which statements to execute.

### Syntax

```
           ┌────────────────────────────────────────────┐
           ▼                                            │
►►──CASE───WHEN──boolean-expression──THEN──statements───┴──ELSE──statements────────►

►─END CASE──────────────────────────────────────────────────────────────────────►◄
```

### Description

**CASE**
A keyword that introduces the first WHEN clause in the CASE statement.

**WHEN** *boolean-expression*
Specifies an expression that is evaluated when control flow enters the WHEN clause in which the expression is defined. If *boolean-expression* evaluates to TRUE, the statements in the corresponding THEN clause are executed. If *boolean-expression* does not evaluate to TRUE, the statements in the corresponding ELSE clause are executed.

**THEN**
A keyword that introduces the statements that are to be executed when the corresponding Boolean expression evaluates to TRUE.

*statements*
Specifies one or more SQL or PL/SQL statements, each terminated with a semicolon.

**ELSE**
A keyword that introduces the default case of the CASE statement.

### Example

The following example uses a searched CASE statement to assign a department name and location to a variable that is based upon the department number.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_loc           dept.loc%TYPE;
    CURSOR emp_cursor IS SELECT empno, ename, deptno FROM emp;
BEGIN
    OPEN emp_cursor;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME     DEPTNO    DNAME     '
        || '     LOC');
    DBMS_OUTPUT.PUT_LINE('-----    -------   ------    ----------'
        || '     ---------');
    LOOP
        FETCH emp_cursor INTO v_empno, v_ename, v_deptno;
        EXIT WHEN emp_cursor%NOTFOUND;
        CASE
            WHEN v_deptno = 10 THEN v_dname := 'Accounting';
                                    v_loc   := 'New York';
            WHEN v_deptno = 20 THEN v_dname := 'Research';
                                    v_loc   := 'Dallas';
            WHEN v_deptno = 30 THEN v_dname := 'Sales';
                                    v_loc   := 'Chicago';
            WHEN v_deptno = 40 THEN v_dname := 'Operations';
                                    v_loc   := 'Boston';
            ELSE v_dname := 'unknown';
                                    v_loc   := '';
        END CASE;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || RPAD(v_ename, 10) ||
            '  ' || v_deptno || '      ' || RPAD(v_dname, 14) || ' ' ||
            v_loc);
    END LOOP;
    CLOSE emp_cursor;
END;
```

This program returns the following sample output:

```
EMPNO    ENAME     DEPTNO    DNAME          LOC
-----    -------   ------    ----------     ---------
7369     SMITH       20      Research       Dallas
7499     ALLEN       30      Sales          Chicago
7521     WARD        30      Sales          Chicago
7566     JONES       20      Research       Dallas
7654     MARTIN      30      Sales          Chicago
7698     BLAKE       30      Sales          Chicago
7782     CLARK       10      Accounting     New York
7788     SCOTT       20      Research       Dallas
7839     KING        10      Accounting     New York
7844     TURNER      30      Sales          Chicago
7876     ADAMS       20      Research       Dallas
7900     JAMES       30      Sales          Chicago
7902     FORD        20      Research       Dallas
7934     MILLER      10      Accounting     New York
```

## Loops (PL/SQL)

Use the EXIT, FOR, LOOP, and WHILE statements to repeat a series of commands in your PL/SQL program.

### FOR (cursor variant) statement (PL/SQL)

The cursor FOR loop statement opens a previously declared cursor, fetches all rows in the cursor result set, and then closes the cursor.

Use this statement instead of separate SQL statements to open a cursor, define a loop construct to retrieve each row of the result set, test for the end of the result set, and then finally close the cursor.

## Invocation

This statement can be invoked within a PL/SQL procedure, function, trigger, or anonymous block.

## Authorization

No specific authorization is required to reference a row expression within an SQL statement; however, for successful statement execution, all other authorization requirements for processing a cursor are required.

## Syntax

```
▶▶──FOR──record──IN──cursor──LOOP──statements──END LOOP──────────────────────▶◀
```

## Description

**FOR**
Introduces the condition that must be true if the FOR loop is to proceed.

*record*
Specifies an identifier that was assigned to an implicitly declared record with definition cursor%ROWTYPE.

**IN** *cursor*
Specifies the name of a previously declared cursor.

**LOOP and END LOOP**
Starts and ends the loop containing SQL statements that are to be executed during each iteration through the loop.

*statements*
One or more PL/SQL statements. A minimum of one statement is required.

## Example

The following example shows a procedure that contains a cursor FOR loop:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
  CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
  DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
  DBMS_OUTPUT.PUT_LINE('-----    -------');
  FOR v_emp_rec IN emp_cur_1 LOOP
    DBMS_OUTPUT.PUT_LINE(v_emp_rec.empno || '     ' || v_emp_rec.ename);
  END LOOP;
END;
```

# FOR (integer variant) statement (PL/SQL)

Use the FOR statement to execute a set of SQL statements more than once.

## Invocation

This statement can be embedded within a PL/SQL procedure, function, or anonymous block statement.

## Authorization

No privileges are required to invoke the FOR statement; however, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded in the FOR statement.

## Syntax

```
►►──FOR──integer-variable──IN──┬─────────┬──expression1──..──expression2──────────►
                               └─REVERSE─┘

►─LOOP──statements──END LOOP─────────────────────────────────────────────────────►◄
```

## Description

*integer-variable*
: An automatically defined integer variable that is used during loop processing. The initial value of *integer-variable* is *expression1*. After the initial iteration, the value of *integer-variable* is incremented at the beginning of each subsequent iteration. Both *expression1* and *expression2* are evaluated when entering the loop, and loop processing stops when *integer-variable* is equal to *expression2*.

**IN** Introduces the optional REVERSE keyword and expressions that define the range of integer variables for the loop.

**REVERSE**
: Specifies that the iteration is to proceed from *expression2* to *expression1*. Note that *expression2* must have a higher value than *expression1*, regardless of whether the REVERSE keyword is specified, if the statements in the loop are to be processed.

*expression1*
: Specifies the initial value of the range of integer variables for the loop. If the REVERSE keyword is specified, *expression1* specifies the end value of the range of integer variables for the loop.

*expression2*
: Specifies the end value of the range of integer variables for the loop. If the REVERSE keyword is specified, *expression2* specifies the initial value of the range of integer variables for the loop.

*statements*
: Specifies the PL/SQL and SQL statements that are executed each time that the loop is processed.

## Examples

The following example shows a basic FOR statement within an anonymous block:

```
BEGIN
  FOR i IN 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

This example generates the following output:

```
Iteration # 1
Iteration # 2
Iteration # 3
Iteration # 4
Iteration # 5
Iteration # 6
Iteration # 7
Iteration # 8
Iteration # 9
Iteration # 10
```

If the start value is greater than the end value, the loop body is not executed at all, but no error is returned, as shown by the following example:

```
BEGIN
  FOR i IN 10 .. 1 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

This example generates no output, because the loop body is never executed.

The following example uses the REVERSE keyword:

```
BEGIN
  FOR i IN REVERSE 1 .. 10 LOOP
    DBMS_OUTPUT.PUT_LINE('Iteration # ' || i);
  END LOOP;
END;
```

This example generates the following output:

```
Iteration # 10
Iteration # 9
Iteration # 8
Iteration # 7
Iteration # 6
Iteration # 5
Iteration # 4
Iteration # 3
Iteration # 2
Iteration # 1
```

## FORALL statement (PL/SQL)

The FORALL statement executes a data change statement for all elements of an array or for a range of elements of an array.

### Invocation

This statement can only be specified in a PL/SQL block.

### Authorization

The privileges held by the authorization ID of the statement must include all of the privileges necessary to invoke the data change statement that is specified in the FORALL statement.

### Syntax

```
►►──FORALL──index-variable──IN──┬─lower-bound──..──upper-bound─┬──────────────────►
                                ├─INDICES OF──indexing-array───┤
                                └─VALUES OF──indexing-array────┘
```

```
►──┬─insert-statement───────────┬────────────────────────────────────────►◄
   ├─searched-delete-statement──┤
   ├─searched-update-statement──┤
   └─execute-immediate-statement┘
```

## Description

*index-variable*
Identifies a name to be used as an array index. It is implicitly declared as an INTEGER and it can only be referenced in the FORALL statement.

*lower-bound* **..** *upper-bound*
Identifies a range of index values that are assignable to the *index-variable* with *lower-bound* less than *upper-bound*. The range represents every integer value starting with *lower-bound* and incrementing by 1 up to and including *upper-bound*.

**INDICES OF** *indexing-array*
Identifies the set of array index values of the array identified by *indexing–array*. If *indexing-array* is an associative array, array index values must be assignable to *index-variable* and could be a sparse set.

**VALUES OF** *indexing-array*
Identifies the set of element values of the array identified by *indexing–array*. The element values must be assignable to *index-variable* and could be an unordered sparse set.

*insert-statement*
Specifies an INSERT statement that is effectively executed for each *index-variable* value.

*searched-delete-statement*
Specifies a searched DELETE statement that is effectively executed for each *index-variable* value.

*searched-update-statement*
Specifies a searched UPDATE statement that is effectively executed for each *index-variable* value.

*execute-immediate-statement*
Specifies an EXECUTE IMMEDIATE statement that is effectively executed for each *index-variable* value.

## Notes

- FORALL statement processing is not atomic. If an error occurs while iterating in the FORALL statement, any data change operations that have already been processed are not implicitly rolled back. An application can use a ROLLBACK statement to roll back the entire transaction when an error occurs in the FORALL statement.

## Example

The following example shows a basic FORALL statement:

```
FORALL x
  IN in_customer_list.FIRST..in_customer_list.LAST
  DELETE FROM customer
    WHERE cust_id IN in_customer_list(x);
```

## EXIT statement (PL/SQL)

The EXIT statement terminates execution of a loop within a PL/SQL code block.

### Invocation

This statement can be embedded within a FOR, LOOP, or WHILE statement, or within a PL/SQL procedure, function, or anonymous block statement.

### Authorization

No privileges are required to invoke the EXIT statement. However, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded within the FOR, LOOP, or WHILE statement.

### Syntax

```
►►──EXIT──────────────────────────────────────────────────────────►◄
```

### Example

The following example shows a basic LOOP statement with an EXIT statement within an anonymous block:

```
DECLARE
  sum PLS_INTEGER := 0;
BEGIN
  LOOP
    sum := sum + 1;
    IF sum > 10 THEN
       EXIT;
    END IF;
  END LOOP;
END
```

## LOOP statement (PL/SQL)

The LOOP statement executes a sequence of statements within a PL/SQL code block multiple times.

### Invocation

This statement can be embedded in a PL/SQL procedure, function, or anonymous block statement.

### Authorization

No privileges are required to invoke the LOOP statement. However, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded within the LOOP statement.

### Syntax

```
►►──LOOP──statements──END──LOOP───────────────────────────────────►◄
```

### Description

*statements*
Specifies one or more PL/SQL or SQL statements. These statements are executed during each iteration of the loop.

### Example

The following example shows a basic LOOP statement within an anonymous block:

```
DECLARE
  sum INTEGER := 0;
BEGIN
  LOOP
    sum := sum + 1;
    IF sum > 10 THEN
       EXIT;
    END IF;
  END LOOP;
END
```

## WHILE statement (PL/SQL)

The WHILE statement repeats a set of SQL statements as long as a specified expression is true. The condition is evaluated immediately before each entry into the loop body.

### Invocation

This statement can be embedded within a PL/SQL procedure, function, or anonymous block statement.

### Authorization

No privileges are required to invoke the WHILE statement; however, the authorization ID of the statement must hold the necessary privileges to invoke the SQL statements that are embedded in the WHILE statement.

### Syntax

```
►►──WHILE──expression──LOOP──statements──END LOOP──────────────────────────────►◄
```

### Description

*expression*
Specifies an expression that is evaluated immediately before each entry into the loop body to determine whether or not the loop is to be executed. If the expression is logically true, the loop is executed. If the expression is logically false, loop processing ends. An EXIT statement can be used to terminate the loop while the expression is true.

*statements*
Specifies the PL/SQL and SQL statements that are executed each time that the loop is processed.

### Example

The following example shows a basic WHILE statement within an anonymous block:

```
DECLARE
  sum INTEGER := 0;
BEGIN
  WHILE sum < 11 LOOP
    sum := sum + 1;
  END LOOP;
END
```

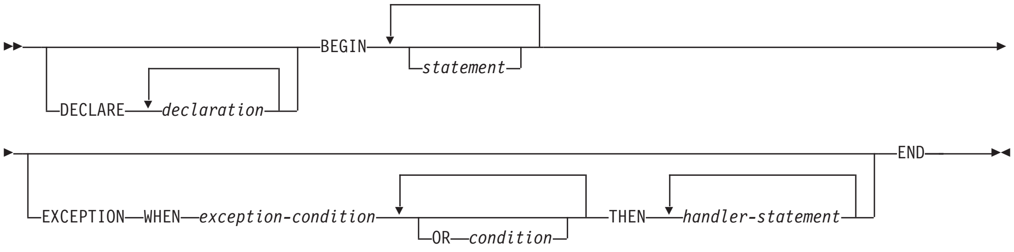
The WHILE statement within this anonymous block executes until *sum* is equal to 11; loop processing then ends, and processing of the anonymous block proceeds to completion.

## Exception handling (PL/SQL)

By default, any error encountered in a PL/SQL program stops execution of the program. You can trap and recover from errors by using an EXCEPTION section.

The syntax for exception handlers is an extension of the syntax for a BEGIN block.

### Syntax



If no error occurs, the block simply executes *statement*, and control passes to the statement after END. But if an error occurs while executing a *statement*, further processing of the *statement* is abandoned, and control passes to the EXCEPTION list. The list is searched for the first *condition* matching the error that occurred. If a match is found, the corresponding *handler-statement* is executed, and control passes to the statement after END. If no match is found, the program stops executing.

If a new error occurs during execution of the *handler-statement*, it can only be caught by a surrounding EXCEPTION clause.

Table 8 summarizes the system-defined conditions that you can use. The special condition name OTHERS matches every error type. Condition names are not case sensitive.

*Table 8. System-defined exception condition names*

| Condition name | Description |
|---|---|
| CASE_NOT_FOUND | None of the cases in a CASE statement evaluates to “true”, and there is no ELSE condition. |
| CURSOR_ALREADY_OPEN | An attempt was made to open a cursor that is already open. |
| DUP_VAL_ON_INDEX | There are duplicate values for the index key. |
| INVALID_CURSOR | An attempt was made to access an unopened cursor. |

*Table 8. System-defined exception condition names (continued)*

| Condition name | Description |
|---|---|
| INVALID_NUMBER | The numeric value is invalid. |
| LOGIN_DENIED | The user name or password is invalid. |
| NO_DATA_FOUND | No rows satisfied the selection criteria. |
| NOT_LOGGED_ON | A database connection does not exist. |
| OTHERS | For any exception that has not been caught by a prior condition in the exception section. |
| SUBSCRIPT_BEYOND_COUNT | An array index is out of range or does not exist. |
| SUBSCRIPT_OUTSIDE_LIMIT | The data type of an array index expression is not assignable to the array index type. |
| TOO_MANY_ROWS | More than one row satisfied the selection criteria, but only one row is allowed to be returned. |
| VALUE_ERROR | The value is invalid. |
| ZERO_DIVIDE | Division by zero was attempted. |

## Raise application error (PL/SQL)

The RAISE_APPLICATION_ERROR procedure makes a user-defined code and error message available to the program which can then be used to identify the exception. This procedure is only supported in PL/SQL contexts.

### Syntax

```
▸▸──RAISE_APPLICATION_ERROR──(──error-number──,──message──)──;──────────────────▸◂
```

### Description

*error-number*
A vendor-specific number (expressed as a literal) that is mapped to a DB2 error code before it is stored in a variable named SQLCODE. The RAISE_APPLICATION_ERROR procedure accepts user-defined *error-number* values from -20000 to -20999. The SQLCODE that is returned in the error message is SQL0438N. The SQLSTATE contains class 'UD' plus three characters that correspond to the last three digits of the *error-number* value.

*message*
A user-defined message with a maximum length of 70 bytes.

### Example

The following example uses the RAISE_APPLICATION_ERROR procedure to display error codes and messages that are specific to missing employee information:

```
CREATE OR REPLACE PROCEDURE verify_emp (
    p_empno         NUMBER
)
IS
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE;
    v_mgr           emp.mgr%TYPE;
    v_hiredate      emp.hiredate%TYPE;
BEGIN
    SELECT ename, job, mgr, hiredate
        INTO v_ename, v_job, v_mgr, v_hiredate FROM emp
        WHERE empno = p_empno;
    IF v_ename IS NULL THEN
        RAISE_APPLICATION_ERROR(-20010, 'No name for ' || p_empno);
    END IF;
    IF v_job IS NULL THEN
        RAISE_APPLICATION_ERROR(-20020, 'No job for' || p_empno);
    END IF;
    IF v_mgr IS NULL THEN
        RAISE_APPLICATION_ERROR(-20030, 'No manager for ' || p_empno);
    END IF;
    IF v_hiredate IS NULL THEN
        RAISE_APPLICATION_ERROR(-20040, 'No hire date for ' || p_empno);
    END IF;
    DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno ||
        ' validated without errors');
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('SQLCODE: ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM: ' || SQLERRM);
END;

CALL verify_emp(7839);

SQLCODE: -438
SQLERRM: SQL0438N  Application raised error or warning with
 diagnostic text: "No manager for 7839".  SQLSTATE=UD030
```

## RAISE statement (PL/SQL)

The RAISE statement raises a previously-defined condition.

### Syntax

```
►►──RAISE──condition──────────────────────────────────────────────────►◄
```

### Description

*condition*
Specifies a previously-defined condition.

### Example

The following example shows a procedure that raises a defined condition:

```
CREATE OR REPLACE PROCEDURE raise_demo (inval NUMBER) IS
  evenno EXCEPTION;
  oddno  EXCEPTION;
BEGIN
  IF MOD(inval, 2) = 1 THEN
    RAISE oddno;
  ELSE
    RAISE evenno;
  END IF;
EXCEPTION
  WHEN evenno THEN
    dbms_output.put_line(TO_CHAR(inval) || ' is even');
  WHEN oddno THEN
    dbms_output.put_line(TO_CHAR(inval) || ' is odd');
END raise_demo;
/
```

```
SET SERVEROUTPUT ON;

CALL raise_demo;
```

## Oracle-DB2 error mapping (PL/SQL)

PL/SQL error codes and exception names have corresponding DB2 error codes and SQLSTATE values.

These error codes, exception names, and SQLSTATE values are summarized in Table 9.

*Table 9. Mapping of PL/SQL error codes and exception names to DB2 error codes and SQLSTATE values*

| plsqlCode | plsqlName | db2Code | db2State |
|---|---|---|---|
| -1 | DUP_VAL_ON_INDEX | -803 | 23505 |
| +100 | NO_DATA_FOUND | +100 | 02000 |
| -1012 | NOT_LOGGED_ON | -1024 | 08003 |
| -1017 | LOGIN_DENIED | -30082 | 08001 |
| -1476 | ZERO_DIVIDE | -801 | 22012 |
| -1722 | INVALID_NUMBER | -420 | 22018 |
| -1001 | INVALID_CURSOR | -501 | 24501 |
| -1422 | TOO_MANY_ROWS | -811 | 21000 |
| -6502 | VALUE_ERROR | -433 | 22001 |
| -6511 | CURSOR_ALREADY_OPEN | -502 | 24502 |
| -6532 | SUBSCRIPT_OUTSIDE_LIMIT | -20438 | 428H1 |
| -6533 | SUBSCRIPT_BEYOND_COUNT | -20439 | 2202E |
| -6592 | CASE_NOT_FOUND | -773 | 20000 |
| -54 | | -904 | 57011 |
| -60 | | -911 | 40001 |
| -310 | | -206 | 42703 |
| -595 | | -390 | 42887 |
| -597 | | -303 | 42806 |
| -598 | | -407 | 23502 |
| -600 | | -30071 | 58015 |
| -603 | | -119 | 42803 |
| -604 | | -119 | 42803 |
| -610 | | -20500 | 428HR |
| -611 | | -117 | 42802 |
| -612 | | -117 | 42802 |
| -613 | | -811 | 21000 |
| -615 | | -420 | 22018 |
| -616 | | -420 | 22018 |
| -617 | | -418 | 42610 |
| -618 | | -420 | 22018 |

*Table 9. Mapping of PL/SQL error codes and exception names to DB2 error codes and SQLSTATE values (continued)*

| plsqlCode | plsqlName | db2Code | db2State |
|---|---|---|---|
| -619 | | -418 | 42610 |
| -620 | | -171 | 42815 |
| -622 | | -304 | 22003 |
| -623 | | -604 | 42611 |
| -904 | | -206 | 42703 |
| -911 | | -7 | 42601 |
| -942 | | -204 | 42704 |
| -955 | | -601 | 42710 |
| -996 | | -1022 | 57011 |
| -1119 | | -292 | 57047 |
| -1002 | | +231 | 02000 |
| -1403 | | -100 | 02000 |
| -1430 | | -612 | 42711 |
| -1436 | | -20451 | 560CO |
| -1438 | | -413 | 22003 |
| -1450 | | -614 | 54008 |
| -1578 | | -1007 | 58034 |
| -2112 | | -811 | 21000 |
| -2261 | | +605 | 01550 |
| -2291 | | -530 | 23503 |
| -2292 | | -532 | 23001 |
| -3113 | | -30081 | 08001 |
| -3114 | | -1024 | 08003 |
| -3214 | | -20170 | 57059 |
| -3297 | | -20170 | 57059 |
| -4061 | | -727 | 56098 |
| -4063 | | -727 | 56098 |
| -4091 | | -723 | 09000 |
| -6502 | | -304 | 22003 |
| -6508 | | -440 | 42884 |
| -6550 | | -104 | 42601 |
| -6553 | | -104 | 42601 |
| -14028 | | -538 | 42830 |
| -19567 | | -1523 | 55039 |
| -30006 | | -904 | 57011 |
| -30041 | | -1139 | 54047 |

# Cursors (PL/SQL)

A *cursor* is a named control structure used by an application program to point to and select a row of data from a result set. Instead of executing a query all at once, you can use a cursor to read and process the query result set one row at a time.

A cursor in a PL/SQL context is treated as a WITH HOLD cursor. For more information about WITH HOLD cursors, see “DECLARE CURSOR statement”.

The DB2 data server supports both PL/SQL static cursors and cursor variables.

## Static cursors (PL/SQL)

A *static cursor* is a cursor whose associated query is fixed at compile time. Declaring a cursor is a prerequisite to using it. Declarations of static cursors using PL/SQL syntax within PL/SQL contexts are supported by the DB2 server.

### Syntax

```
►►──CURSOR──cursor-name──IS──query──────────────────────────────────►◄
```

### Description

*cursor-name*
Specifies an identifier for the cursor that can be used to reference the cursor and its result set.

*query*
Specifies a SELECT statement that determines a result set for the cursor.

### Example

The following example shows a procedure that contains multiple static cursor declarations:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
     CURSOR emp_cur_1 IS SELECT * FROM emp;

     CURSOR emp_cur_2 IS SELECT empno, ename FROM emp;

     CURSOR emp_cur_3 IS SELECT empno, ename
                         FROM emp
                         WHERE deptno = 10
                         ORDER BY empno;
BEGIN
    OPEN emp_cur_1;
        ...
END;
```

### Parameterized cursors (PL/SQL)

Parameterized cursors are static cursors that can accept passed-in parameter values when they are opened.

The following example includes a parameterized cursor. The cursor displays the name and salary of each employee in the EMP table whose salary is less than that specified by a passed-in parameter value.

```
DECLARE
    my_record       emp%ROWTYPE;
    CURSOR c1 (max_wage NUMBER) IS
```

```
        SELECT * FROM emp WHERE sal < max_wage;
BEGIN
    OPEN c1(2000);
    LOOP
        FETCH c1 INTO my_record;
        EXIT WHEN c1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name = ' || my_record.ename || ', salary = '
            || my_record.sal);
    END LOOP;
    CLOSE c1;
END;
```

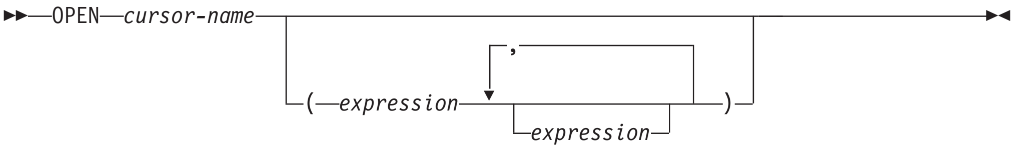
If 2000 is passed in as the value of *max_wage*, only the name and salary data for those employees whose salary is less than 2000 is returned:

```
Name = SMITH, salary = 800.00
Name = ALLEN, salary = 1600.00
Name = WARD, salary = 1250.00
Name = MARTIN, salary = 1250.00
Name = TURNER, salary = 1500.00
Name = ADAMS, salary = 1100.00
Name = JAMES, salary = 950.00
Name = MILLER, salary = 1300.00
```

## Opening a cursor (PL/SQL)

The result set that is associated with a cursor cannot be referenced until the cursor has been opened.

### Syntax



### Description

*cursor-name*
Specifies an identifier for a cursor that was previously declared within a PL/SQL context. The specified cursor cannot already be open.

*expression*
When *cursor-name* is a parameterized cursor, specifies one or more optional actual parameters. The number of actual parameters must match the number of corresponding formal parameters.

### Example

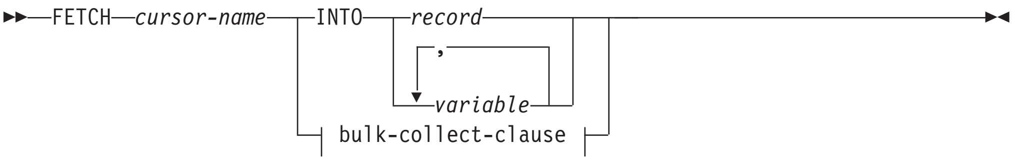
The following example shows an OPEN statement for a cursor that is part of the CURSOR_EXAMPLE procedure:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    CURSOR emp_cur_3 IS SELECT empno, ename
                        FROM emp
                        WHERE deptno = 10
                        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
        ...
END;
```

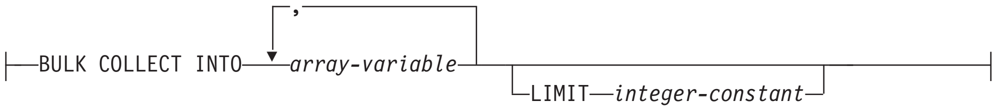
## Fetching rows from a cursor (PL/SQL)

The FETCH statement that is required to fetch rows from a PL/SQL cursor is supported by the DB2 data server in PL/SQL contexts.

### Syntax



#### bulk-collect-clause:



### Description

*cursor-name*
: Name of a static cursor or cursor variable.

*record*
: Identifier for a previously-defined record. This can be a user-defined record or a record definition that is derived from a table using the %ROWTYPE attribute.

*variable*
: A PL/SQL variable that will hold the field data from the fetched row. One or more variables can be defined, but they must match in order and number the fields that are returned in the select list of the query that was specified in the cursor declaration. The data types of the fields in the select list must match or be implicitly convertible to the data types of the fields in the record or the data types of the variables.

The variable data types can be defined explicitly or by using the %TYPE attribute.

**BULK COLLECT INTO** *array-variable*
: Identifies one or more variables with an array data type. Each row of the result is assigned to an element in each array in the order of the result set, with the array index assigned in sequence.

- If exactly one *array-variable* is specified:
  - If the data type of the *array-variable* element is not a record type, the result row of the cursor must have exactly one column, and the column data type must be assignable to the array element data type.
  - If the data type of the *array-variable* element is a record type, the result row of the cursor must be assignable to the record type.
- If multiple array variables are specified:
  - The data type of the *array-variable* element must not be a record type.
  - There must be an *array-variable* for each column in the result row of the cursor.

- The data type of each column in the result row of the cursor must be assignable to the array element data type of the corresponding *array-variable*.

If the data type of *array-variable* is an ordinary array, the maximum cardinality must be greater than or equal to the number of rows that are returned by the query, or greater than or equal to the *integer-constant* that is specified in the LIMIT clause.

**LIMIT** *integer-constant*
Identifies a limit for the number of rows stored in the target array. The cursor position is moved forward *integer-constant* rows or to the end of the result set.

## Example

The following example shows a procedure that contains a FETCH statement.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         NUMBER(4);
    v_ename         VARCHAR2(10);
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

If the %TYPE attribute is used to define the data type of a target variable, the target variable declaration in a PL/SQL application program does not need to change if the data type of the database column changes. The following example shows a procedure with variables that are defined using the %TYPE attribute.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    CURSOR emp_cur_3 IS SELECT empno, ename FROM emp WHERE deptno = 10
        ORDER BY empno;
BEGIN
    OPEN emp_cur_3;
    FETCH emp_cur_3 INTO v_empno, v_ename;
        ...
END;
```

If all of the columns in a table are retrieved in the order in which they are defined, the %ROWTYPE attribute can be used to define a record into which the FETCH statement will place the retrieved data. Each field within the record can then be accessed using dot notation. The following example shows a procedure with a record definition that uses %ROWTYPE. This record is used as the target of the FETCH statement.

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
     v_emp_rec       emp%ROWTYPE;
     CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
     OPEN emp_cur_1;
     FETCH emp_cur_1 INTO v_emp_rec;
     DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
     DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
END;
```

## Closing a cursor (PL/SQL)

After all rows have been retrieved from the result set that is associated with a cursor, the cursor must be closed. The result set cannot be referenced after the cursor has been closed.

However, the cursor can be reopened and the rows of the new result set can be fetched.

### Syntax

```
►►──CLOSE──cursor-name──────────────────────────────────────────────────────►◄
```

### Description

*cursor-name*
: Specifies an identifier for an open cursor that was previously declared within a PL/SQL context.

### Example

The following example shows a CLOSE statement for a cursor that is part of the CURSOR_EXAMPLE procedure:

```
CREATE OR REPLACE PROCEDURE cursor_example
IS
    v_emp_rec       emp%ROWTYPE;
    CURSOR emp_cur_1 IS SELECT * FROM emp;
BEGIN
    OPEN emp_cur_1;
    FETCH emp_cur_1 INTO v_emp_rec;
    DBMS_OUTPUT.PUT_LINE('Employee Number: ' || v_emp_rec.empno);
    DBMS_OUTPUT.PUT_LINE('Employee Name  : ' || v_emp_rec.ename);
    CLOSE emp_cur_1;
END;
```

## Using %ROWTYPE with cursors (PL/SQL)

The %ROWTYPE attribute is used to define a record with fields corresponding to all of the columns that are fetched from a cursor or cursor variable. Each field assumes the data type of its corresponding column.

The %ROWTYPE attribute is prefixed by a cursor name or a cursor variable name. The syntax is `record cursor%ROWTYPE`, where *record* is an identifier that is assigned to the record, and *cursor* is an explicitly declared cursor within the current scope.

The following example shows how to use a cursor with the %ROWTYPE attribute to retrieve department information about each employee in the EMP table.

```
CREATE OR REPLACE PROCEDURE emp_info
IS
    CURSOR empcur IS SELECT ename, deptno FROM emp;
    myvar           empcur%ROWTYPE;
BEGIN
    OPEN empcur;
    LOOP
        FETCH empcur INTO myvar;
        EXIT WHEN empcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE( myvar.ename || ' works in department '
            || myvar.deptno );
    END LOOP;
    CLOSE empcur;
END;
```

A call to this procedure (`CALL emp_info;`) returns the following sample output:

```
SMITH works in department 20
ALLEN works in department 30
WARD works in department 30
JONES works in department 20
MARTIN works in department 30
BLAKE works in department 30
CLARK works in department 10
SCOTT works in department 20
KING works in department 10
TURNER works in department 30
ADAMS works in department 20
JAMES works in department 30
FORD works in department 20
MILLER works in department 10
```

## Cursor attributes (PL/SQL)

Each cursor has a set of attributes that enables an application program to test the state of the cursor.

These attributes are %ISOPEN, %FOUND, %NOTFOUND, and %ROWCOUNT.

**%ISOPEN**
This attribute is used to determine whether a cursor is in the open state. When a cursor is passed as a parameter to a function or procedure, it is useful to know (before attempting to open the cursor) whether the cursor is already open.

**%FOUND**
This attribute is used to determine whether a cursor contains rows after the execution of a FETCH statement. If FETCH statement execution was successful, the %FOUND attribute has a value of true. If FETCH statement execution was not successful, the %FOUND attribute has a value of false. The result is unknown when:

- The value of *cursor-variable-name* is null
- The underlying cursor of *cursor-variable-name* is not open
- The %FOUND attribute is evaluated before the first FETCH statement was executed against the underlying cursor
- FETCH statement execution returns an error

The %FOUND attribute provides an efficient alternative to using a condition handler that checks for the error that is returned when no more rows remain to be fetched.

**%NOTFOUND**
This attribute is the logical opposite of the %FOUND attribute.

**%ROWCOUNT**
This attribute is used to determine the number of rows that have been fetched since a cursor was opened.

Table 10 summarizes the attribute values that are associated with certain cursor events.

*Table 10. Summary of cursor attribute values*

| Cursor attribute | %ISOPEN | %FOUND | %NOTFOUND | %ROWCOUNT |
|---|---|---|---|---|
| Before OPEN | False | Undefined | Undefined | “Cursor not open” exception |

Table 10. Summary of cursor attribute values (continued)

| Cursor attribute | %ISOPEN | %FOUND | %NOTFOUND | %ROWCOUNT |
|---|---|---|---|---|
| After OPEN and before 1st FETCH | True | Undefined | Undefined | 0 |
| After 1st successful FETCH | True | True | False | 1 |
| After *n*th successful FETCH (last row) | True | True | False | *n* |
| After *n*+1st FETCH (after last row) | True | False | True | *n* |
| After CLOSE | False | Undefined | Undefined | "Cursor not open" exception |

# Cursor variables (PL/SQL)

A *cursor variable* is a cursor that contains a pointer to a query result set. The result set is determined by execution of the OPEN FOR statement using the cursor variable.

A cursor variable, unlike a static cursor, is not associated with a particular query. The same cursor variable can be opened a number of times with separate OPEN FOR statements containing different queries. A new result set is created each time and made available through the cursor variable.

## SYS_REFCURSOR cursor variables (PL/SQL)

The DB2 server supports the declaration of cursor variables of the SYS_REFCURSOR built-in data type, which can be associated with any result set.

The SYS_REFCURSOR data type is known as a weakly-typed REF CURSOR type. Strongly-typed cursor variables of the REF CURSOR type require a result set specification.

### Syntax

```
►►──DECLARE──cursor-variable-name──SYS_REFCURSOR──────────────────────────────►◄
```

### Description

*cursor-variable-name*
Specifies an identifier for the cursor variable.

**SYS_REFCURSOR**
Specifies that the data type of the cursor variable is the system-defined SYS_REFCURSOR data type.

### Example

The following example shows a SYS_REFCURSOR variable declaration:

```
DECLARE emprefcur SYS_REFCURSOR;
```

## User-defined REF CURSOR type variables (PL/SQL)

The DB2 server supports the user-defined REF CURSOR data type and cursor variable declarations.

The user-defined REF CURSOR type can be defined by executing the TYPE declaration in a PL/SQL context. After the type has been defined, you can declare a cursor variable of that type.

### Syntax

```
►►──TYPE──cursor-type-name──IS REF CURSOR──┬─────────────────────────┬──────────►◄
                                           └─RETURN──return-type─┘
```

### Description

**TYPE** *cursor-type-name*
Specifies an identifier for the cursor data type.

**IS REF CURSOR**
Specifies that the cursor is of a user-defined REF CURSOR data type.

**RETURN** *return-type*
Specifies the return type that is associated with the cursor. If a *return-type* is specified, this REF CURSOR type is strongly typed; otherwise, it is weakly typed.

### Example

The following example shows a cursor variable declaration in the DECLARE section of an anonymous block:

```
DECLARE
    TYPE emp_cur_type IS REF CURSOR RETURN emp%ROWTYPE;
    my_rec emp_cur_type;
BEGIN
    ...
END
```

## Dynamic queries with cursor variables (PL/SQL)

The DB2 data server supports dynamic queries through the OPEN FOR statement in PL/SQL contexts.

### Syntax

```
►►──OPEN──cursor-variable-name──FOR──dynamic-string──┬──────────────────────┬──────►◄
                                                     │         ┌─,───────┐  │
                                                     │         ▼         │  │
                                                     └─USING────bind-arg─┴──┘
```

### Description

**OPEN** *cursor-variable-name*
Specifies an identifier for a cursor variable that was previously declared within a PL/SQL context.

**FOR** *dynamic-string*
Specifies a string literal or string variable that contains a SELECT statement

(without the terminating semicolon). The statement can contain named parameters, such as, for example, *:param1*.

**USING** *bind-arg*
Specifies one or more bind arguments whose values are substituted for placeholders in *dynamic-string* when the cursor opens.

## Examples

The following example shows a dynamic query that uses a string literal:

```
CREATE OR REPLACE PROCEDURE dept_query
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = 30' ||
        ' AND sal >= 1500';
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following example output is generated by the DEPT_QUERY procedure:

```
CALL dept_query;

EMPNO    ENAME
-----    -------
7499     ALLEN
7698     BLAKE
7844     TURNER
```

The query in the previous example can be modified with bind arguments to pass the query parameters:

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
    OPEN emp_refcur FOR 'SELECT empno, ename FROM emp WHERE deptno = :dept'
        || ' AND sal >= :sal' USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

The following CALL statement generates the same output that was generated in the previous example:

```
CALL dept_query(30, 1500);
```

A string variable to pass the SELECT statement provides the most flexibility:

```
CREATE OR REPLACE PROCEDURE dept_query (
    p_deptno        emp.deptno%TYPE,
    p_sal           emp.sal%TYPE
)
IS
    emp_refcur      SYS_REFCURSOR;
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    p_query_string  VARCHAR2(100);
BEGIN
    p_query_string := 'SELECT empno, ename FROM emp WHERE ' ||
        'deptno = :dept AND sal >= :sal';
    OPEN emp_refcur FOR p_query_string USING p_deptno, p_sal;
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH emp_refcur INTO v_empno, v_ename;
        EXIT WHEN emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE emp_refcur;
END;
```

This version of the DEPT_QUERY procedure generates the following example output:

```
CALL dept_query(20, 1500);

EMPNO    ENAME
-----    -------
7566     JONES
7788     SCOTT
7902     FORD
```

## Example: Returning a REF CURSOR from a procedure (PL/SQL)

This example demonstrates how to define and open a REF CURSOR variable, and then pass it as a procedure parameter.

The cursor variable is specified as an IN OUT parameter so that the result set is made available to the caller of the procedure:

```
CREATE OR REPLACE PROCEDURE emp_by_job (
    p_job           VARCHAR2,
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp WHERE job = p_job;
END;
```

The EMP_BY_JOB procedure is invoked in the following anonymous block by assigning the procedure's IN OUT parameter to a cursor variable that was declared in the anonymous block's declaration section. The result set is fetched using this cursor variable.

```
DECLARE
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
    v_job           emp.job%TYPE := 'SALESMAN';
    v_emp_refcur    SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES WITH JOB ' || v_job);
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    emp_by_job(v_job, v_emp_refcur);
    LOOP
        FETCH v_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN v_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
    CLOSE v_emp_refcur;
END;
```

The following example output is generated when the anonymous block executes:

```
EMPLOYEES WITH JOB SALESMAN
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7844     TURNER
```

## Example: Modularizing cursor operations (PL/SQL)

This example demonstrates how various operations on cursor variables can be modularized into separate programs or PL/SQL components.

The following example shows a procedure that opens a cursor variable whose query retrieves all rows in the EMP table:

```
CREATE OR REPLACE PROCEDURE open_all_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp;
END;
```

In the next example, a procedure opens a cursor variable whose query retrieves all rows for a given department:

```
CREATE OR REPLACE PROCEDURE open_emp_by_dept (
    p_emp_refcur    IN OUT SYS_REFCURSOR,
    p_deptno        emp.deptno%TYPE
)
IS
BEGIN
    OPEN p_emp_refcur FOR SELECT empno, ename FROM emp
        WHERE deptno = p_deptno;
END;
```

The following example shows a procedure that opens a cursor variable whose query retrieves all rows in the DEPT table:

```
CREATE OR REPLACE PROCEDURE open_dept (
    p_dept_refcur    IN OUT SYS_REFCURSOR
)
IS
BEGIN
    OPEN p_dept_refcur FOR SELECT deptno, dname FROM dept;
END;
```

In the next example, a procedure fetches and displays a cursor variable result set consisting of employee number and name:

```
CREATE OR REPLACE PROCEDURE fetch_emp (
    p_emp_refcur    IN OUT SYS_REFCURSOR
)
IS
    v_empno         emp.empno%TYPE;
    v_ename         emp.ename%TYPE;
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH p_emp_refcur INTO v_empno, v_ename;
        EXIT WHEN p_emp_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_empno || '     ' || v_ename);
    END LOOP;
END;
```

The following example shows a procedure that fetches and displays a cursor variable result set consisting of department number and name:

```
CREATE OR REPLACE PROCEDURE fetch_dept (
    p_dept_refcur   IN SYS_REFCURSOR
)
IS
    v_deptno        dept.deptno%TYPE;
    v_dname         dept.dname%TYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('DEPT   DNAME');
    DBMS_OUTPUT.PUT_LINE('----   ---------');
    LOOP
        FETCH p_dept_refcur INTO v_deptno, v_dname;
        EXIT WHEN p_dept_refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(v_deptno || '     ' || v_dname);
    END LOOP;
END;
```

The following example shows a procedure that closes a cursor variable:

```
CREATE OR REPLACE PROCEDURE close_refcur (
    p_refcur        IN OUT SYS_REFCURSOR
)
IS
BEGIN
    CLOSE p_refcur;
END;
```

The following example shows an anonymous block that executes these procedures:

```
DECLARE
    gen_refcur      SYS_REFCURSOR;
BEGIN
    DBMS_OUTPUT.PUT_LINE('ALL EMPLOYEES');
    open_all_emp(gen_refcur);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('****************');

    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #10');
    open_emp_by_dept(gen_refcur, 10);
    fetch_emp(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('****************');

    DBMS_OUTPUT.PUT_LINE('DEPARTMENTS');
    open_dept(gen_refcur);
    fetch_dept(gen_refcur);
    DBMS_OUTPUT.PUT_LINE('*****************');

    close_refcur(gen_refcur);
END;
```

The following example output is generated when the anonymous block executes:

```
ALL EMPLOYEES
EMPNO    ENAME
-----    -------
7369     SMITH
7499     ALLEN
```

```
7521     WARD
7566     JONES
7654     MARTIN
7698     BLAKE
7782     CLARK
7788     SCOTT
7839     KING
7844     TURNER
7876     ADAMS
7900     JAMES
7902     FORD
7934     MILLER
****************
EMPLOYEES IN DEPT #10
EMPNO    ENAME
-----    -------
7782     CLARK
7839     KING
7934     MILLER
****************
DEPARTMENTS
DEPT   DNAME
----   ---------
10     ACCOUNTING
20     RESEARCH
30     SALES
40     OPERATIONS
*****************
```

# Triggers (PL/SQL)

A PL/SQL trigger is a named database object that encapsulates and defines a set of actions that are to be performed in response to an insert, update, or delete operation against a table. Triggers are created using the PL/SQL CREATE TRIGGER statement.

## Types of triggers (PL/SQL)

The DB2 data server supports row-level triggers within a PL/SQL context.

A *row-level trigger* fires once for each row that is affected by a triggering event. For example, if deletion is defined as a triggering event for a particular table, and a single DELETE statement deletes five rows from that table, the trigger will fire five times, once for each row.

The trigger code block is executed either before or after each row is affected by the triggering statement.

## Trigger variables (PL/SQL)

NEW and OLD are special variables that you can use with PL/SQL triggers without explicitly defining them.

- NEW is a pseudo-record name that refers to the new table row for insert and update operations in row-level triggers. Its usage is `:NEW.column`, where *column* is the name of a column in the table on which the trigger is defined.
  - When used in a *before row-level trigger*, the initial content of :NEW.*column* is the column value in the new row that is to be inserted or in the row that is to replace the old row.
  - When used in an *after row-level trigger*, the new column value has already been stored in the table.

In the trigger code block, :NEW.*column* can be used like any other variable. If a value is assigned to :NEW.*column* in the code block of a before row-level trigger, the assigned value is used in the inserted or updated row.

- OLD is a pseudo-record name that refers to the old table row for update and delete operations in row-level triggers. Its usage is `:OLD.column`, where *column* is the name of a column in the table on which the trigger is defined.
  - When used in a *before row-level trigger*, the initial content of :OLD.*column* is the column value in the row that is to be deleted or in the old row that is to be replaced by the new row.
  - When used in an *after row-level trigger*, the old column value is no longer stored in the table.

  In the trigger code block, :OLD.*column* can be used like any other variable. If a value is assigned to :OLD.*column* in the code block of a before row-level trigger, the assigned value has no affect on the action of the trigger.

## Transactions and exceptions (PL/SQL)

A trigger is always executed as part of the same transaction within which the triggering statement is executing.

If no exceptions occur within the trigger code block, the effects of data manipulation language (DML) within the trigger are committed only if the transaction that contains the triggering statement commits. If the transaction is rolled back, the effects of DML within the trigger are also rolled back.

A DB2 rollback can only occur within an atomic block or by using an UNDO handler. The triggering statement itself is not rolled back unless the application forces a rollback of the encapsulating transaction.

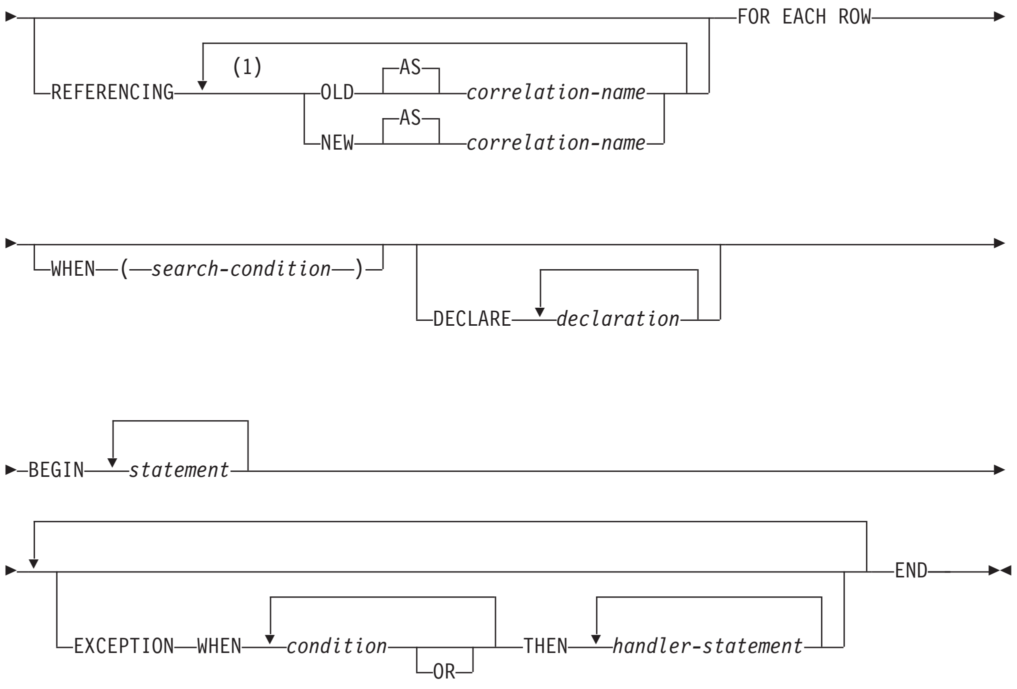
If an unhandled exception occurs within the trigger code block, the calling statement is rolled back.

## CREATE TRIGGER statement (PL/SQL)

The CREATE TRIGGER statement defines a PL/SQL trigger in the database.

### Syntax

```
►►──CREATE──┬────────────┬──TRIGGER──trigger-name──┬─BEFORE─┬──────────────────►
            └─OR REPLACE─┘                         └─AFTER──┘


►──┬─INSERT─────────────────────────┬──ON──table-name──────────────────────────►
   ├─DELETE─────────────────────────┤
   └─UPDATE──┬───────────────────┬──┘
             │      ┌─,────────┐ │
             │      ▼          │ │
             └─OF───┴─column-name┴─┘
```



**Notes:**

1 OLD and NEW can only be specified once each.

## Description

**OR REPLACE**
Specifies to replace the definition for the trigger if one exists at the current server. The existing definition is effectively dropped before the new definition is replaced in the catalog. This option is ignored if a definition for the trigger does not exist at the current server.

*trigger-name*
Names the trigger. The name, including the implicit or explicit schema name, must not identify a trigger already described in the catalog (SQLSTATE 42710). If a two-part name is specified, the schema name cannot begin with 'SYS' (SQLSTATE 42939).

**BEFORE**
Specifies that the associated triggered action is to be applied before any changes caused by the actual update of the subject table are applied to the database. It also specifies that the triggered action of the trigger will not cause other triggers to be activated.

**AFTER**
Specifies that the associated triggered action is to be applied after the changes caused by the actual update of the subject table are applied to the database.

**INSERT**
Specifies that the triggered action associated with the trigger is to be executed whenever an INSERT operation is applied to the subject table.

**DELETE**
Specifies that the triggered action associated with the trigger is to be executed whenever a DELETE operation is applied to the subject table.

**UPDATE**
Specifies that the triggered action associated with the trigger is to be executed whenever an UPDATE operation is applied to the subject table, subject to the columns specified or implied.

If the optional *column-name* list is not specified, every column of the table is implied. Therefore, omission of the *column-name* list implies that the trigger will be activated by the update of any column of the table.

**OF** *column-name*,...
Each *column-name* specified must be a column of the base table (SQLSTATE 42703). If the trigger is a BEFORE trigger, the *column-name* specified cannot be a generated column other than the identity column (SQLSTATE 42989). No *column-name* can appear more than once in the *column-name* list (SQLSTATE 42711). The trigger will only be activated by the update of a column that is identified in the *column-name* list.

**ON** *table-name*
Designates the subject table of the BEFORE trigger or AFTER trigger definition. The name must specify a base table or an alias that resolves to a base table (SQLSTATE 42704 or 42809). The name must not specify a catalog table (SQLSTATE 42832), a materialized query table (SQLSTATE 42997), a created temporary table, a declared temporary table (SQLSTATE 42995), or a nickname (SQLSTATE 42809).

**REFERENCING**
Specifies the correlation names for the *transition variables* and the table names for the *transition tables*. Correlation names identify a specific row in the set of rows affected by the triggering SQL operation. Table names identify the complete set of affected rows. Each row affected by the triggering SQL operation is available to the triggered action by qualifying columns with *correlation-names* specified as follows.

**OLD AS** *correlation-name*
Specifies a correlation name that identifies the row state prior to the triggering SQL operation.

**NEW AS** *correlation-name*
Specifies a correlation name that identifies the row state as modified by the triggering SQL operation and by any SET statement in a BEFORE trigger that has already executed.

**FOR EACH ROW**
Specifies that the triggered action is to be applied once for each row of the subject table that is affected by the triggering SQL operation.

**WHEN**

**(***search-condition***)**
Specifies a condition that is true, false, or unknown. The *search-condition* provides a capability to determine whether or not a certain triggered action should be executed. The associated action is performed only if the specified search condition evaluates as true.

*declaration*
Specifies a variable declaration.

*statement* **or** *handler-statement*
Specifies a PL/SQL program statement. The trigger body can contain nested blocks.

*condition*
: Specifies an exception condition name, such as NO_DATA_FOUND.

### Example

The following example shows a before row-level trigger that calculates the commission of every new employee belonging to department 30 before a record for that employee is inserted into the EMP table:

```
CREATE TABLE emp (
    name            VARCHAR2(10),
    deptno          NUMBER,
    sal             NUMBER,
    comm            NUMBER
)
/

CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :NEW.deptno = 30 THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;
END
/
```

## Dropping triggers (PL/SQL)

You can remove a trigger from the database by using the DROP TRIGGER statement.

### Syntax

```
►►──DROP TRIGGER──trigger-name─────────────────────────────────────────────►◄
```

### Description

*trigger-name*
: Specifies the name of the trigger that is to be dropped.

## Examples: Triggers (PL/SQL)

PL/SQL trigger definitions can be compiled by the DB2 data server. These examples will help you to create valid triggers and to troubleshoot PL/SQL trigger compilation errors.

### Before row-level triggers

The following example shows a before row-level trigger that calculates the commission of every new employee belonging to department 30 before a record for that employee is inserted into the EMP table:

```
CREATE OR REPLACE TRIGGER emp_comm_trig
    BEFORE INSERT ON emp
    FOR EACH ROW
BEGIN
    IF :NEW.deptno = 30 THEN
        :NEW.comm := :NEW.sal * .4;
    END IF;
END;
```

The trigger computes the commissions for two new employees and inserts those values as part of the new employee rows:

```
INSERT INTO emp VALUES (9005,'ROBERS','SALESMAN',7782,SYSDATE,3000.00,NULL,30);

INSERT INTO emp VALUES (9006,'ALLEN','SALESMAN',7782,SYSDATE,4500.00,NULL,30);

SELECT * FROM emp WHERE empno IN (9005, 9006);

     EMPNO ENAME      JOB              MGR HIREDATE         SAL       COMM     DEPTNO
---------- ---------- --------- ---------- --------- ---------- ---------- ----------
      9005 ROBERS     SALESMAN        7782 01-APR-05       3000       1200         30
      9006 ALLEN      SALESMAN        7782 01-APR-05       4500       1800         30
```

## After row-level triggers

The following example shows three after row-level triggers.

- When a new employee row is inserted into the EMP table, one trigger (EMP_INS_TRIG) adds a new row to the JOBHIST table for that employee and adds a row to the EMPCHGLOG table with a description of the action.
- When an existing employee row is updated, the second trigger (EMP_CHG_TRIG) sets the ENDDATE column of the latest JOBHIST row (assumed to be the one with a null ENDDATE) to the current date and inserts a new JOBHIST row with the employee's new information. This trigger also adds a row to the EMPCHGLOG table with a description of the action
- When an employee row is deleted from the EMP table, the third trigger (EMP_DEL_TRIG) adds a row to the EMPCHGLOG table with a description of the action.

```
CREATE TABLE empchglog (
    chg_date        DATE,
    chg_desc        VARCHAR2(30)
);
CREATE OR REPLACE TRIGGER emp_ins_trig
    AFTER INSERT ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Added';
    v_empno := :NEW.empno;
    v_deptno := :NEW.deptno;
    INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
        :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, 'New Hire');

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;

CREATE OR REPLACE TRIGGER emp_chg_trig
    AFTER UPDATE ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Updated';
    v_empno := :NEW.empno;
    v_deptno := :NEW.deptno;
    v_chgdesc := '';
    IF NVL(:OLD.ename, '-null-') != NVL(:NEW.ename, '-null-') THEN
        v_chgdesc := v_chgdesc || 'name, ';
    END IF;
    IF NVL(:OLD.job, '-null-') != NVL(:NEW.job, '-null-') THEN
        v_chgdesc := v_chgdesc || 'job, ';
    END IF;
    IF NVL(:OLD.sal, -1) != NVL(:NEW.sal, -1) THEN
        v_chgdesc := v_chgdesc || 'salary, ';
    END IF;
    IF NVL(:OLD.comm, -1) != NVL(:NEW.comm, -1) THEN
        v_chgdesc := v_chgdesc || 'commission, ';
    END IF;
    IF NVL(:OLD.deptno, -1) != NVL(:NEW.deptno, -1) THEN
        v_chgdesc := v_chgdesc || 'department, ';
    END IF;
    v_chgdesc := 'Changed ' || RTRIM(v_chgdesc, ', ');
    UPDATE jobhist SET enddate = SYSDATE WHERE empno = :OLD.empno
        AND enddate IS NULL;
    INSERT INTO jobhist VALUES (:NEW.empno, SYSDATE, NULL,
        :NEW.job, :NEW.sal, :NEW.comm, :NEW.deptno, v_chgdesc);

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;

CREATE OR REPLACE TRIGGER emp_del_trig
    AFTER DELETE ON emp
    FOR EACH ROW
DECLARE
    v_empno         emp.empno%TYPE;
    v_deptno        emp.deptno%TYPE;
    v_dname         dept.dname%TYPE;
    v_action        VARCHAR2(7);
    v_chgdesc       jobhist.chgdesc%TYPE;
BEGIN
    v_action := 'Deleted';
    v_empno := :OLD.empno;
    v_deptno := :OLD.deptno;

    INSERT INTO empchglog VALUES (SYSDATE,
        v_action || ' employee # ' || v_empno);
END;
```

In the following example, two employee rows are added using two separate INSERT statements, and then both rows are updated using a single UPDATE statement. The JOBHIST table shows the action of the trigger for each affected row: two new hire entries for the two new employees and two changed commission records. The EMPCHGLOG table also shows that the trigger was fired a total of four times, once for each action against the two rows.

```
INSERT INTO emp VALUES (9003,'PETERS','ANALYST',7782,SYSDATE,5000.00,NULL,40);

INSERT INTO emp VALUES (9004,'AIKENS','ANALYST',7782,SYSDATE,4500.00,NULL,40);

UPDATE emp SET comm = sal * 1.1 WHERE empno IN (9003, 9004);

SELECT * FROM jobhist WHERE empno IN (9003, 9004);

     EMPNO STARTDATE ENDDATE   JOB              SAL       COMM     DEPTNO CHGDESC
---------- --------- --------- --------- ---------- ---------- ---------- ------------------
      9003 31-MAR-05 31-MAR-05 ANALYST         5000                    40 New Hire
      9004 31-MAR-05 31-MAR-05 ANALYST         4500                    40 New Hire
      9003 31-MAR-05           ANALYST         5000       5500         40 Changed commission
      9004 31-MAR-05           ANALYST         4500       4950         40 Changed commission

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
```

```
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
```

After both employees are deleted with a single DELETE statement, the EMPCHGLOG table shows that the trigger was fired twice, once for each deleted employee:

```
DELETE FROM emp WHERE empno IN (9003, 9004);

SELECT * FROM empchglog;

CHG_DATE  CHG_DESC
--------- ------------------------------
31-MAR-05 Added employee # 9003
31-MAR-05 Added employee # 9004
31-MAR-05 Updated employee # 9003
31-MAR-05 Updated employee # 9004
31-MAR-05 Deleted employee # 9003
31-MAR-05 Deleted employee # 9004
```

# Packages (PL/SQL)

PL/SQL package definitions are supported by the DB2 data server. A PL/SQL package is a named collection of functions, procedures, variables, cursors, user-defined types, and records that are referenced using a common qualifier, the package name.

Packages have the following characteristics:

- Packages provide a convenient way of organizing the functions and procedures that have a related purpose. Permission to use the package functions and procedures is dependent upon one privilege that is granted to the entire package.
- Certain items in a package can be declared public. Public entities are visible and can be referenced by other programs that hold the EXECUTE privilege on the package. In the case of public functions and procedures, only their signatures are visible. The PL/SQL code for these function and procedures is not accessible to others; therefore, applications that utilize such a package are dependent upon only the information that is available in the signatures.
- Other items in a package can be declared private. Private entities can be referenced and used by functions and procedures within the package, but not by external applications.

## Package components (PL/SQL)

Packages consist of two main components: the package specification and the package body.

- The *package specification* is the public interface, comprising the elements that can be referenced outside of the package. A package specification is created by executing the CREATE PACKAGE statement.
- The *package body* contains the actual implementation of all of the procedures and functions that are declared within the package specification, as well as any declaration of private types, variables, and cursors. A package body is created by executing the CREATE PACKAGE BODY statement.

## Creating packages (PL/SQL)

Creating a package specification enables you to encapsulate related data type, procedure, and function definitions within a single context in the database.

Packages are extensions of schemas that provide namespace support for the objects that they reference. They are repositories in which executable code can be defined. Using a package involves referencing or executing objects that are defined in the package specification and implemented within the package.

## Creating package specifications (PL/SQL)

A package specification establishes which package objects can be referenced from outside of the package. Objects that can be referenced from outside of a package are called the public elements of that package.

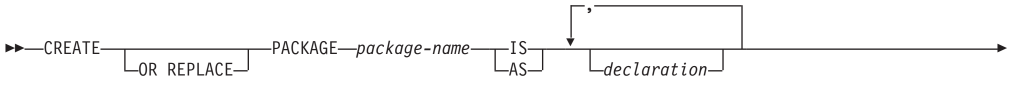
The following example shows how to create a package specification named EMP_ADMIN, consisting of two functions and two stored procedures.

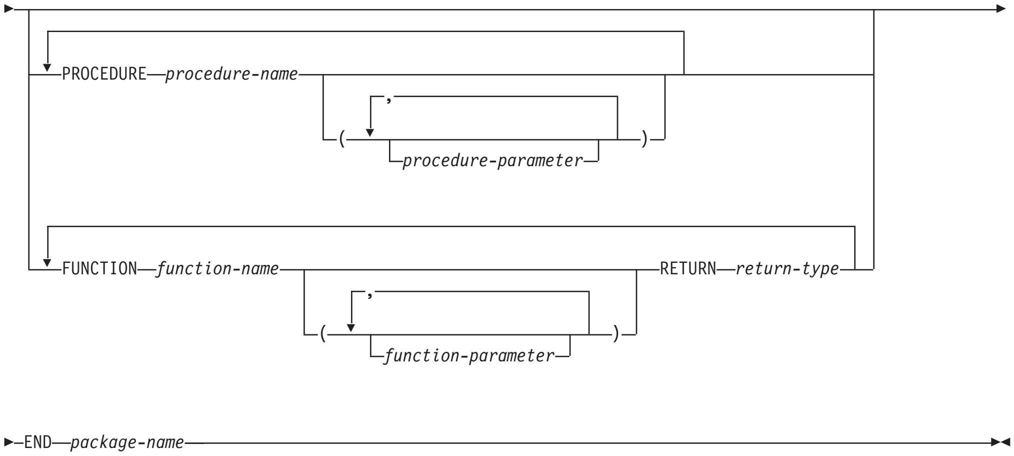
```
CREATE OR REPLACE PACKAGE emp_admin
IS

   FUNCTION get_dept_name (
      p_deptno        NUMBER DEFAULT 10
   )
   RETURN VARCHAR2;
   FUNCTION update_emp_sal (
      p_empno         NUMBER,
      p_raise         NUMBER
   )
   RETURN NUMBER;
   PROCEDURE hire_emp (
      p_empno         NUMBER,
      p_ename         VARCHAR2,
      p_job           VARCHAR2,
      p_sal           NUMBER,
      p_hiredate      DATE DEFAULT sysdate,
      p_comm          NUMBER DEFAULT 0,
      p_mgr           NUMBER,
      p_deptno        NUMBER DEFAULT 10
   );
   PROCEDURE fire_emp (
      p_empno         NUMBER
   );

END emp_admin;
```

## CREATE PACKAGE statement (PL/SQL)

The CREATE PACKAGE statement creates a package specification, which defines the interface to a package.

### Syntax





## Description

*package-name*
Specifies an identifier for the package.

*declaration*
Specifies an identifier for a public item. The public item can be accessed from outside of the package using the syntax `package-name.item-name`. There can be zero or more public items. Public item declarations must come before procedure or function declarations. The *declaration* can be any of the following:

- Collection declaration
- EXCEPTION declaration
- Record declaration
- REF CURSOR and cursor variable declaration
- TYPE definition for a collection, record, or REF CURSOR type variable
- Variable declaration

*procedure-name*
Specifies an identifier for a public procedure. The public procedure can be invoked from outside of the package using the syntax `package-name.procedure-name()`.

*procedure-parameter*
Specifies an identifier for a formal parameter of the procedure.

*function-name*
Specifies an identifier for a public function. The public function can be invoked from outside of the package using the syntax `package-name.function-name()`.

*function-parameter*
Specifies an identifier for a formal parameter of the function. Input (IN mode) parameters can be initialized with a default value.

*return-type*
Specifies a data type for the value that is returned by the function.

## Notes

The CREATE PACKAGE statement can be submitted in obfuscated form. In an obfuscated statement, only the package name is readable. The rest of the statement is encoded in such a way that it is not readable, but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

## Creating the package body (PL/SQL)

A package body contains the implementation of all of the procedures and functions that are declared within the package specification.

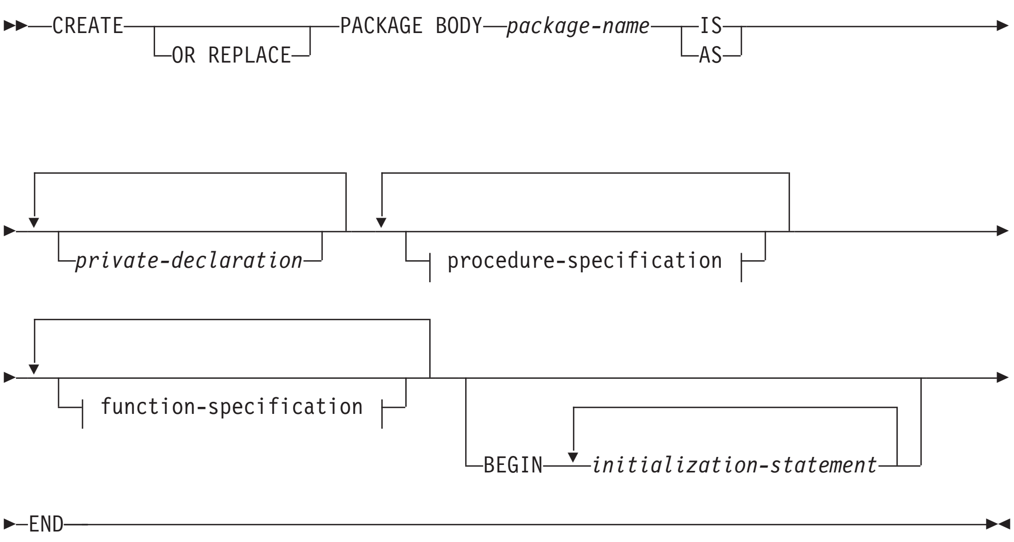
The following example shows how to create a package body for the EMP_ADMIN package specification.

```
--
--  Package body for the 'emp_admin' package.
--
CREATE OR REPLACE PACKAGE BODY emp_admin
IS
   --
   --  Function that queries the 'dept' table based on the department
   --  number and returns the corresponding department name.
   --
   FUNCTION get_dept_name (
      p_deptno        IN NUMBER DEFAULT 10
   )
   RETURN VARCHAR2
   IS
      v_dname         VARCHAR2(14);
   BEGIN
      SELECT dname INTO v_dname FROM dept WHERE deptno = p_deptno;
      RETURN v_dname;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Invalid department number ' || p_deptno);
         RETURN '';
   END;
   --
   --  Function that updates an employee's salary based on the
   --  employee number and salary increment/decrement passed
   --  as IN parameters.  Upon successful completion the function
   --  returns the new updated salary.
   --
   FUNCTION update_emp_sal (
      p_empno         IN NUMBER,
      p_raise         IN NUMBER
   )
   RETURN NUMBER
   IS
      v_sal           NUMBER := 0;
   BEGIN
      SELECT sal INTO v_sal FROM emp WHERE empno = p_empno;
      v_sal := v_sal + p_raise;
      UPDATE emp SET sal = v_sal WHERE empno = p_empno;
      RETURN v_sal;
   EXCEPTION
      WHEN NO_DATA_FOUND THEN
         DBMS_OUTPUT.PUT_LINE('Employee ' || p_empno || ' not found');
         RETURN -1;
      WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('The following is SQLERRM:');
         DBMS_OUTPUT.PUT_LINE(SQLERRM);
         DBMS_OUTPUT.PUT_LINE('The following is SQLCODE:');
         DBMS_OUTPUT.PUT_LINE(SQLCODE);
         RETURN -1;
   END;
   --
   --  Procedure that inserts a new employee record into the 'emp' table.
   --
   PROCEDURE hire_emp (
      p_empno         NUMBER,
      p_ename         VARCHAR2,
      p_job           VARCHAR2,
      p_sal           NUMBER,
      p_hiredate      DATE    DEFAULT sysdate,
      p_comm          NUMBER  DEFAULT 0,
      p_mgr           NUMBER,
      p_deptno        NUMBER  DEFAULT 10
   )
   AS
   BEGIN
      INSERT INTO emp(empno, ename, job, sal, hiredate, comm, mgr, deptno)
         VALUES(p_empno, p_ename, p_job, p_sal,
                p_hiredate, p_comm, p_mgr, p_deptno);
   END;
   --
   --  Procedure that deletes an employee record from the 'emp' table based
   --  on the employee number.
   --
   PROCEDURE fire_emp (
      p_empno         NUMBER
   )
   AS
   BEGIN
      DELETE FROM emp WHERE empno = p_empno;
   END;
END;
```

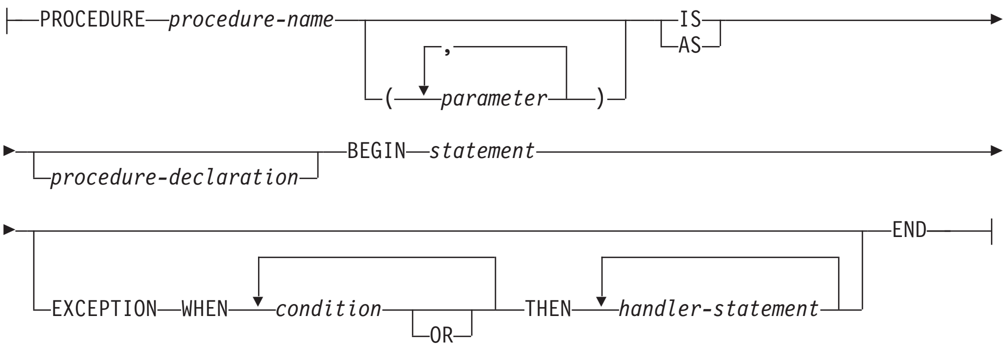
## CREATE PACKAGE BODY statement (PL/SQL)

The CREATE PACKAGE BODY statement creates a package body, which contains the implementation of all of the procedures and functions that are declared within the package specification, as well as any declaration of private types, variables, and cursors.

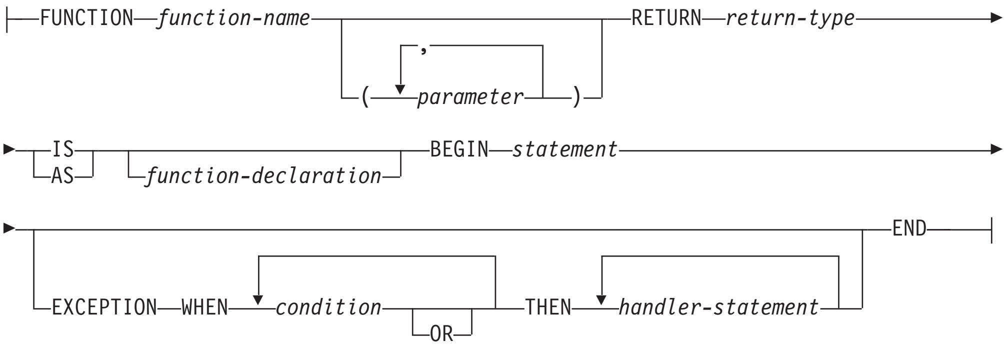
### Syntax



**procedure-specification:**



**function-specification:**



## Description

*package-name*
Specifies the name of the package whose body is to be created. A package specification with the same name must exist.

*private-declaration*
Specifies the name of a private variable that can be accessed by any procedure or function within the package. There can be zero or more private variables. The *private-declaration* can be any of the following:

- Variable declaration
- Record declaration
- Collection declaration
- REF CURSOR and cursor variable declaration
- TYPE definitions for records, collections, or variables of the REF CURSOR type

*procedure-name*
Specifies the name of a public procedure that is declared in the package specification and its signature. The signature can specify any one of the following: the formal parameter names, data types, parameter modes, the order of the formal parameters, or the number of formal parameters. When the procedure name and package specification exactly match the signature of the public procedure's declaration, *procedure-name* defines the body of this public procedure.

If none of these conditions is true, *procedure-name* defines a new private procedure.

*parameter*
Specifies a formal parameter of the procedure.

*procedure-declaration*
Specifies a declaration that can be accessed only from within procedure *procedure-name*. This is a PL/SQL statement.

*statement*
Specifies a PL/SQL program statement.

*function-name*
Specifies the name of a public function that is declared in the package specification and its signature. The signature can specify any one of the following: the formal parameter names, data types, parameter modes, the order of the formal parameters, or the number of formal parameters. When the function name and package specification exactly match the signature of the public function's declaration, *function-name* defines the body of this public function.

If none of these conditions is true, *function-name* defines a new private function.

*parameter*
Specifies a formal parameter of the function.

*return-type*
Specifies the data type of the value that is returned by the function.

*function-declaration*
Specifies a declaration that can be accessed only from within function *function-name*. This is a PL/SQL statement.

*statement*
Specifies a PL/SQL program statement.

*initialization-statement*
Specifies a statement in the initialization section of the package body. The initialization section, if specified, must contain at least one statement. The statements in the initialization section are executed once per user session when the package is first referenced.

### Notes

The CREATE PACKAGE BODY statement can be submitted in obfuscated form. In an obfuscated statement, only the package name is readable. The rest of the statement is encoded in such a way that it is not readable, but can be decoded by the database server. Obfuscated statements can be produced by calling the DBMS_DDL.WRAP function.

## Referencing package objects (PL/SQL)

References to objects that are defined within a package must sometimes be qualified with the package name.

To reference the objects that are declared in a package specification, specify the package name, a period character, and then the name of the object. If the package is not defined in the current schema, specify the schema name as well. For example:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
schema.package_name.subprogram_name
```

## Example

The following example contains a reference to a function named GET_DEPT_NAME that is defined in a package named EMP_ADMIN:

```
select emp_admin.get_dept_name(10) from dept
```

## Packages with user-defined types (PL/SQL)

User-defined types can be declared and referenced in packages.

The following example shows a package specification for the EMP_RPT package. This definition includes the following declarations:

- A publicly accessible record type, EMPREC_TYP
- A publicly accessible weakly-typed REF CURSOR type, EMP_REFCUR
- Two functions, GET_DEPT_NAME and OPEN_EMP_BY_DEPT; the latter function returns the REF CURSOR type EMP_REFCUR
- Two procedures, FETCH_EMP and CLOSE_REFCUR; both declare a weakly-typed REF CURSOR type as a formal parameter

```
CREATE OR REPLACE PACKAGE emp_rpt
IS
    TYPE emprec_typ IS RECORD (
        empno       NUMBER(4),
        ename       VARCHAR(10)
    );
    TYPE emp_refcur IS REF CURSOR;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2;
    FUNCTION open_emp_by_dept (
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR;
    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    );
    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    );
END emp_rpt;
```

The definition of the associated package body includes the following private variable declarations:

- A static cursor, DEPT_CUR
- An associative array type, DEPTTAB_TYP
- An associative array variable, T_DEPT
- An integer variable, T_DEPT_MAX
- A record variable, R_EMP

```
CREATE OR REPLACE PACKAGE BODY emp_rpt
IS
    CURSOR dept_cur IS SELECT * FROM dept;
    TYPE depttab_typ IS TABLE of dept%ROWTYPE
        INDEX BY BINARY_INTEGER;
    t_dept          DEPTTAB_TYP;
    t_dept_max      INTEGER := 1;
    r_emp           EMPREC_TYP;

    FUNCTION get_dept_name (
        p_deptno    IN NUMBER
    ) RETURN VARCHAR2
    IS
    BEGIN
        FOR i IN 1..t_dept_max LOOP
            IF p_deptno = t_dept(i).deptno THEN
                RETURN t_dept(i).dname;
            END IF;
        END LOOP;
        RETURN 'Unknown';
    END;

    FUNCTION open_emp_by_dept(
        p_deptno    IN emp.deptno%TYPE
    ) RETURN EMP_REFCUR
    IS
        emp_by_dept EMP_REFCUR;
    BEGIN
        OPEN emp_by_dept FOR SELECT empno, ename FROM emp
            WHERE deptno = p_deptno;
        RETURN emp_by_dept;
    END;

    PROCEDURE fetch_emp (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
        DBMS_OUTPUT.PUT_LINE('-----    -------');
        LOOP
            FETCH p_refcur INTO r_emp;
            EXIT WHEN p_refcur%NOTFOUND;
            DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' || r_emp.ename);
        END LOOP;
    END;

    PROCEDURE close_refcur (
        p_refcur    IN OUT SYS_REFCURSOR
    )
    IS
    BEGIN
        CLOSE p_refcur;
    END;
BEGIN
    OPEN dept_cur;
    LOOP
        FETCH dept_cur INTO t_dept(t_dept_max);
        EXIT WHEN dept_cur%NOTFOUND;
        t_dept_max := t_dept_max + 1;
    END LOOP;
    CLOSE dept_cur;
    t_dept_max := t_dept_max - 1;
END emp_rpt;
```

This package contains an initialization section that loads the private associative array variable T_DEPT, using the private static cursor DEPT_CUR. T_DEPT serves as a department name lookup table in function GET_DEPT_NAME. The function OPEN_EMP_BY_DEPT returns a REF CURSOR variable for the result set of employee numbers and names for a given department. This REF CURSOR variable can then be passed to procedure FETCH_EMP to retrieve and list the individual rows of the result set. Finally, procedure CLOSE_REFCUR can be used to close the REF CURSOR variable that is associated with this result set.

The following anonymous block runs the package functions and procedures. The declaration section includes the declaration of cursor variable V_EMP_CUR, using the public REF CURSOR type, EMP_REFCUR. V_EMP_CUR contains a pointer to the result set that is passed between the package function and procedures.

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    emp_rpt.fetch_emp(v_emp_cur);
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    emp_rpt.close_refcur(v_emp_cur);
END;
```

This anonymous block produces the following sample output:

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

The following anonymous block shows another way of achieving the same result. Instead of using the package procedures FETCH_EMP and CLOSE_REFCUR, the logic is coded directly into the anonymous block. Note the declaration of record variable R_EMP, using the public record type EMPREC_TYP.

```
DECLARE
    v_deptno        dept.deptno%TYPE DEFAULT 30;
    v_emp_cur       emp_rpt.EMP_REFCUR;
    r_emp           emp_rpt.EMPREC_TYP;
BEGIN
    v_emp_cur := emp_rpt.open_emp_by_dept(v_deptno);
    DBMS_OUTPUT.PUT_LINE('EMPLOYEES IN DEPT #' || v_deptno ||
        ': ' || emp_rpt.get_dept_name(v_deptno));
    DBMS_OUTPUT.PUT_LINE('EMPNO    ENAME');
    DBMS_OUTPUT.PUT_LINE('-----    -------');
    LOOP
        FETCH v_emp_cur INTO r_emp;
        EXIT WHEN v_emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(r_emp.empno || '     ' ||
            r_emp.ename);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('**********************');
    DBMS_OUTPUT.PUT_LINE(v_emp_cur%ROWCOUNT || ' rows were retrieved');
    CLOSE v_emp_cur;
END;
```

This anonymous block produces the following sample output:

```
EMPLOYEES IN DEPT #30: SALES
EMPNO    ENAME
-----    -------
```

```
7499     ALLEN
7521     WARD
7654     MARTIN
7698     BLAKE
7844     TURNER
7900     JAMES
**********************
6 rows were retrieved
```

## Dropping packages (PL/SQL)

You can drop a package if it is no longer needed. Alternatively, if you want to reuse the package, you have the option to drop only the package body.

### Syntax

```
►►──DROP──PACKAGE──┬──────┬──package-name──────────────────────────────────►◄
                   └─BODY─┘
```

### Description

**BODY**
Specifies that only the package body is to be dropped. If this keyword is omitted, both the package specification and the package body are dropped.

*package-name*
Specifies the name of a package.

### Examples

The following example shows how to drop only the body of a package named EMP_ADMIN:

```
DROP PACKAGE BODY emp_admin
```

The following example shows how to drop both the specification and the body of the package:

```
DROP PACKAGE emp_admin
```

# Chapter 3. System-defined modules

The system-defined modules provide an easy-to-use programmatic interface for performing a variety of useful operations.

For example, you can use system-defined modules to perform the following functions:

- Send and receive messages and alerts across connections.
- Write to and read from files and directories on the operating system's file system.
- Generate reports containing a variety of monitor information.

System-defined modules can be invoked from an SQL-based application, a DB2 command line, or a command script.

System-defined modules are not supported for the following product editions:

- DB2 Express®
- DB2 Express-C
- DB2 Personal Edition

## DBMS_ALERT module

The DBMS_ALERT module provides a set of procedures for registering for alerts, sending alerts, and receiving alerts.

Alerts are stored in SYSTOOLS.DBMS_ALERT_INFO, which is created in the SYSTOOLSPACE when you first reference this module for each database.

The schema for this module is SYSIBMADM.

The DBMS_ALERT module includes the following system-defined routines.

*Table 11. System-defined routines available in the DBMS_ALERT module*

| Routine name | Description |
|---|---|
| REGISTER procedure | Registers the current session to receive a specified alert. |
| REMOVE procedure | Removes registration for a specified alert. |
| REMOVEALL procedure | Removes registration for all alerts. |
| SIGNAL procedure | Signals the occurrence of a specified alert. |
| SET_DEFAULTS procedure | Sets the polling interval for the WAITONE and WAITANY procedures. |
| WAITANY procedure | Waits for any registered alert to occur. |
| WAITONE procedure | Waits for a specified alert to occur. |

### Usage notes

The procedures in the DBMS_ALERT module are useful when you want to send an alert for a specific event. For example, you might want to send an alert when a trigger is activated as the result of changes to one or more tables.

© Copyright IBM Corp. 1993, 2010

The DBMS_ALERT module requires that the database configuration parameter CUR_COMMIT is set to ON

### Example

When a trigger, TRIG1, is activated, send an alert from connection 1 to connection 2 . First, create the table and the trigger.

```
CREATE TABLE T1 (C1 INT)@

CREATE TRIGGER TRIG1
AFTER INSERT ON T1
REFERENCING NEW AS NEW
FOR EACH ROW
BEGIN ATOMIC
CALL DBMS_ALERT.SIGNAL( 'trig1', NEW.C1 );
END@
```

From connection 1, issue an INSERT statement.

```
INSERT INTO T1 values (10)@
-- Commit to send messages to the listeners (required in early program)
CALL DBMS_ALERT.COMMIT()@
```

From connection 2, register to receive the alert called trig1 and wait for the alert.

```
CALL DBMS_ALERT.REGISTER('trig1')@
CALL DBMS_ALERT.WAITONE('trig1', ?, ?, 5)@
```

This example results in the following output:

```
  Value of output parameters
  --------------------------
  Parameter Name  : MESSAGE
  Parameter Value : -

  Parameter Name  : STATUS
  Parameter Value : 1

  Return Status = 0
```

## REGISTER procedure - Register to receive a specified alert

The REGISTER procedure registers the current session to receive a specified alert.

### Syntax

```
►►—DBMS_ALERT.REGISTER—(—name—)————————————————————————————————►◄
```

### Procedure parameters

*name*
An input argument of type VARCHAR(128) that specifies the name of the alert.

### Authorization

EXECUTE privilege on the DBMS_ALERT module.

### Example

Use the REGISTER procedure to register for an alert named alert_test, and then wait for the signal.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 5;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

This example results in the following output:

```
Waiting for signal...
Alert name   : alert_test
Alert status : 1
```

## REMOVE procedure - Remove registration for a specified alert

The REMOVE procedure removes registration from the current session for a specified alert.

### Syntax

```
►►—DBMS_ALERT.REMOVE—(—name—)————————————————————————————————►◄
```

### Procedure parameters

*name*

An input argument of type VARCHAR(128) that specifies the name of the alert.

### Authorization

EXECUTE privilege on the DBMS_ALERT module.

### Example

Use the REMOVE procedure to remove an alert named alert_test.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 5;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name   : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

This example results in the following output:

```
Waiting for signal...
Alert name   : alert_test
Alert status : 1
```

## REMOVEALL procedure - Remove registration for all alerts

The REMOVEALL procedure removes registration from the current session for all alerts.

### Syntax

```
►►──DBMS_ALERT.REMOVEALL──────────────────────────────────────────────────►◄
```

### Authorization

EXECUTE privilege on the DBMS_ALERT module.

### Example

Use the REMOVEALL procedure to remove registration for all alerts.

```
  CALL DBMS_ALERT.REMOVEALL@
```

## SET_DEFAULTS - Set the polling interval for WAITONE and WAITANY

The SET_DEFAULTS procedure sets the polling interval that is used by the WAITONE and WAITANY procedures.

### Syntax

```
►►──DBMS_ALERT.SET_DEFAULTS──(──sensitivity──)────────────────────────────►◄
```

### Procedure parameters

*sensitivity*
: An input argument of type INTEGER that specifies an interval in seconds for the WAITONE and WAITANY procedures to check for signals. If a value is not specified, then the interval is 1 second by default.

### Authorization

EXECUTE privilege on the DBMS_ALERT module.

### Example

Use the SET_DEFAULTS procedure to specify the polling interval for the WAITONE and WAITANY procedures.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 20;
  DECLARE v_polling INTEGER DEFAULT 3;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.SET_DEFAULTS(v_polling);
  CALL DBMS_OUTPUT.PUT_LINE('Polling interval: ' || v_polling);
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

This example results in the following output:

```
Polling interval : 3
```

## SIGNAL procedure - Signal occurrence of a specified alert

The SIGNAL procedure signals the occurrence of a specified alert. The signal includes a message that is passed with the alert. The message is distributed to the listeners (processes that have registered for the alert) when the SIGNAL call is issued.

### Syntax

```
►►──DBMS_ALERT.SIGNAL──(──name──,──message──)──────────────────────────────►◄
```

### Procedure parameters

*name*
An input argument of type VARCHAR(128) that specifies the name of the alert.

*message*
An input argument of type VARCHAR(32672) that specifies the information to pass with this alert. This message can be returned by the WAITANY or WAITONE procedures when an alert occurs.

### Authorization

EXECUTE privilege on the DBMS_ALERT module.

### Example

Use the SIGNAL procedure to signal the occurrence of an alert named alert_test.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_name   VARCHAR(30) DEFAULT 'alert_test';
  CALL DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END@

CALL proc1@
```

This example results in the following output:

```
Issued alert for alert_test
```

## WAITANY procedure - Wait for any registered alerts

The WAITANY procedure waits for any registered alerts to occur.

## Syntax

```
►►──DBMS_ALERT.WAITANY──(──name──,──message──,──status──,──timeout──)──────────►◄
```

## Procedure parameters

*name*
An output argument of type VARCHAR(128) that contains the name of the alert.

*message*
An output argument of type VARCHAR(32672) that contains the message sent by the SIGNAL procedure.

*status*
An output argument of type INTEGER that contains the status code returned by the procedure. The following values are possible

*0* An alert occurred.

*1* A timeout occurred.

*timeout*
An input argument of type INTEGER that specifies the amount of time in seconds to wait for an alert.

## Authorization

EXECUTE privilege on the DBMS_ALERT module.

## Example

From one connection, run a CLP script called `waitany.clp` to receive any registered alerts.

```
waitany.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30);
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 20;
  CALL DBMS_ALERT.REGISTER('alert_test');
  CALL DBMS_ALERT.REGISTER('any_alert');
  CALL DBMS_OUTPUT.PUT_LINE('Registered for alert alert_test and any_alert');
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITANY(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert msg : ' || v_msg);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
  CALL DBMS_ALERT.REMOVEALL;
END@

call proc1@
```

From another connection, run a script called `signal.clp` to issue a signal for an alert named any_alert.

```
signal.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc2
  BEGIN
  DECLARE v_name VARCHAR(30) DEFAULT 'any_alert';
  CALL DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END@

CALL proc2@
```

The script `signal.clp` results in the following output:

```
Issued alert for any_alert
```

The script `waitany.clp` results in the following output:

```
Registered for alert alert_test and any_alert
Waiting for signal...
Alert name : any_alert
Alert msg : This is the message from any_alert
Alert status : 0
Alert timeout: 20 seconds
```

### Usage notes

If no alerts are registered when the WAITANY procedure is called, the procedure returns SQL0443N.

## WAITONE procedure - Wait for a specified alert

The WAITONE procedure waits for a specified alert to occur.

### Syntax

```
►►──DBMS_ALERT.WAITONE──(──name──,──message──,──status──,──timeout──)──────────►◄
```

### Procedure parameters

*name*
An input argument of type VARCHAR(128) that specifies the name of the alert.

*message*
An output argument of type VARCHAR(32672) that contains the message sent by the SIGNAL procedure.

*status*
An output argument of type INTEGER that contains the status code returned by the procedure. The following values are possible

*0* An alert occurred.

*1* A timeout occurred.

*timeout*
An input argument of type INTEGER that specifies the amount of time in seconds to wait for the specified alert.

### Authorization

EXECUTE privilege on the DBMS_ALERT module.

## Example

Run a CLP script named `waitone.clp` to receive an alert named alert_test.

```
waitone.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_name    VARCHAR(30) DEFAULT 'alert_test';
  DECLARE v_msg     VARCHAR(80);
  DECLARE v_status  INTEGER;
  DECLARE v_timeout INTEGER DEFAULT 20;
  CALL DBMS_ALERT.REGISTER(v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Waiting for signal...');
  CALL DBMS_ALERT.WAITONE(v_name , v_msg , v_status , v_timeout);
  CALL DBMS_OUTPUT.PUT_LINE('Alert name : ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Alert msg : ' || v_msg);
  CALL DBMS_OUTPUT.PUT_LINE('Alert status : ' || v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Alert timeout: ' || v_timeout || ' seconds');
  CALL DBMS_ALERT.REMOVE(v_name);
END@

CALL proc1@
```

From a different connection, run a script named `signalalert.clp` to issue a signal for an alert named alert_test.

```
signalalert.clp:

SET SERVEROUTPUT ON@

CREATE PROCEDURE proc2
BEGIN
  DECLARE v_name VARCHAR(30) DEFAULT 'alert_test';
  CALL DBMS_ALERT.SIGNAL(v_name,'This is the message from ' || v_name);
  CALL DBMS_OUTPUT.PUT_LINE('Issued alert for ' || v_name);
END@

CALL proc2@
```

The script `signalalert.clp` results in the following output:

```
Issued alert for alert_test
```

The script `waitone.clp` results in the following output:

```
Waiting for signal...
Alert name : alert_test
Alert msg : This is the message from alert_test
Alert status : 0
Alert timeout: 20 seconds
```

# DBMS_DDL Module

The DBMS_DDL module provides the capability to obfuscate DDL objects such as routines, triggers, views or PL/SQL packages. Obfuscation allows the deployment of SQL objects to a database without exposing the procedural logic.

The DDL statements for these objects are obfuscated both in vendor-provided install scripts as well as in the DB2 catalogs.

The schema for this module is SYSIBMADM.

*Table 12. The DBMS_DDL module includes the following routines.*

| Routine name | Description |
|---|---|
| WRAP function | Produces an obfuscated version of the DDL statement provided as argument. |
| CREATE_WRAPPED procedure | Deploys a DDL statement in the database in an obfuscated format. |

## WRAP function – Obfuscate a DDL statement

The WRAP function transforms a readable DDL statement into an obfuscated DDL statement.

### Syntax

In an obfuscated DDL statement, the procedural logic and embedded SQL statements are scrambled in such a way that any intellectual property in the logic cannot be easily extracted. If the DDL statement corresponds to an external routine definition, the portion following the parameter list is encoded.

```
►►──WRAP──(──object-definition-string──)─────────────────────────────►◄
```

### Parameters

**object-definition-string**

A string of type CLOB(2M) containing a DDL statement text which can be one of the following (SQLSTATE 5UA0O):

- create procedure
- create function
- create package (PL/SQL)
- create package body (PL/SQL)
- create trigger
- create view
- alter module add function
- alter module publish function
- alter module add procedure
- alter module publish procedure

The result is a string of type CLOB(2M) which contains an encoded version of the input statement. The encoding consists of a prefix of the original statement up to and including the routine signature or the trigger, view or package name, followed by the keyword WRAPPED. This keyword is followed by information about the application server that executes the function. The information has the form *pppvvrrm*, where:

- *ppp* identifies the product as DB2 Database for Linux®, UNIX®, and Windows® using the letters SQL
- *vv* is a two-digit version identifier, such as '09'
- *rr* is a two-digit release identifier, such as '07'
- *m* is a one-character modification level identifier, such as '0'.

For example, Fixpack 2 of Version 9.7 is identified as 'SQL09072'. This application server information is followed by a string of letters (a-z, and A-Z), digits (0-9), underscores and colons. No syntax checking is done on the input statement beyond the prefix that remains readable after obfuscation.

The encoded DDL statement is typically longer than the plain text form of the statement. If the result exceeds the maximum length for SQL statements an error is raised (SQLSTATE 54001).

**Note:** The encoding of the statement is meant to obfuscate the content and should not be considered as a form of strong encryption.

### Authorization

EXECUTE privilege on the DBMS_DDL module

### Example

1. Produce an obfuscated version of a function that computes a yearly salary from an hourly wage given a 40 hour workweek

```
VALUES(DDL.WRAP('CREATE FUNCTION ' ||
                'salary(wage DECFLOAT) ' ||
                'RETURNS DECFLOAT ' ||
                'RETURN wage * 40 * 52'))
The result of the previous statement would be something of the form:

CREATE FUNCTION salary(wage DECFLOAT) WRAPPED SQL09072 obfuscated-text
```

2. Produce an obfuscated form of a trigger setting a complex default

```
VALUES(DBMS_DDL.WRAP('CREATE OR REPLACE TRIGGER ' ||
                     'trg1 BEFORE INSERT ON emp ' ||
                     'REFERENCING NEW AS n ' ||
                     'FOR EACH ROW ' ||
                     'WHEN (n.bonus IS NULL) ' ||
                     'SET n.bonus = n.salary * .04'))

The result of the previous statement would be something of the form:

CREATE OR REPLACE TRIGGER trg1 WRAPPED SQL09072 obfuscated-text
```

## CREATE_WRAPPED procedure – Deploy an obfuscated object

The CREATE_WRAPPED procedure transforms a plain text DDL object definition into an obfuscated DDL object definition and then deploys the object in the database.

### Syntax

In an obfuscated DDL statement, the procedural logic and embedded SQL statements are encoded in such a way that any intellectual property in the logic cannot be easily extracted.

```
►►──CREATE_WRAPPED──(──object-definition-string──)─────────────────────────►◄
```

### Parameters

**object-definition-string**
A string of type CLOB(2M) containing a DDL statement text which can be one of the following (SQLSTATE 5UA0O):

- create procedure

- create function
- create package (PL/SQL)
- create package body (PL/SQL)
- create trigger
- create view
- alter module add function
- alter module publish function
- alter module add procedure
- alter module publish procedure

The procedure transforms the input into an obfuscated DDL statement string and then dynamically executes that DDL statement. Special register values such as PATH and CURRENT SCHEMA in effect at invocation as well as the current invoker's rights are being used.

The encoding consists of a prefix of the original statement up to and including the routine signature or the trigger, view or package name, followed by the keyword WRAPPED. This keyword is followed by information about the application server that executes the procedure. The information has the form "*pppvvrrm*," where:

- *ppp* identifies the product as DB2 Database for Linux®, UNIX®, and Windows® using the letters SQL
- *vv* is a two-digit version identifier, such as '09'
- *rr* is a two-digit release identifier, such as '07'
- *m* is a one-character modification level identifier, such as '0'.

For example, Fixpack 2 of Version 9.7 is identified as 'SQL09072'. This application server information is followed by a string of letters (a-z, and A-Z), digits (0-9), underscores and colons. No syntax checking is done on the input statement beyond the prefix that remains readable after obfuscation.

The encoded DDL statement is typically longer than the plain text form of the statement. If the result exceeds the maximum length for SQL statements an error is raised (SQLSTATE 54001).

**Note:** The encoding of the statement is meant to obfuscate the content and should not be considered as a form of strong encryption.

## Authorization

EXECUTE privilege on the DBMS_DDL module.

## Example

1. Create an obfuscated function computing a yearly salary from an hourly wage given a 40 hour workweek

```
CALL DBMS_DDL.CREATE_WRAPPED('CREATE FUNCTION ' ||
                   'salary(wage DECFLOAT) ' ||
                   'RETURNS DECFLOAT ' ||
                   'RETURN wage * 40 * 52');
SELECT text FROM SYSCAT.ROUTINES
 WHERE routinename = 'SALARY'
    AND routineschema = CURRENT SCHEMA;
```

Upon successful execution of the CALL statement, The SYSCAT.ROUTINES.TEXT column for the row corresponding to routine ‘SALARY&apos; would be something of the form:

```
CREATE FUNCTION salary(wage DECFLOAT) WRAPPED SQL09072 obfuscated-text
```

2. Create an obfuscated trigger setting a complex default

```
CALL DBMS_DDL.CREATE_WRAPPED('CREATE OR REPLACE TRIGGER ' ||
                   'trg1 BEFORE INSERT ON emp ' ||
                   'REFERENCING NEW AS n ' ||
                   'FOR EACH ROW ' ||
                   'WHEN (n.bonus IS NULL) ' ||
                   'SET n.bonus = n.salary * .04');
SELECT text FROM SYSCAT.TRIGGERS
 WHERE trigname = 'TRG1'
   AND trigschema = CURRENT SCHEMA;
```

Upon successful execution of the CALL statement, The SYSCAT.TRIGGERS.TEXT column for the row corresponding to trigger 'TRG1' would be something of the form:

```
CREATE OR REPLACE TRIGGER trg1 WRAPPED SQL09072  obfuscated-text
```

## DBMS_JOB module

The DBMS_JOB module provides procedures for the creation, scheduling, and managing of jobs.

The DBMS_JOB module provides an alternate interface for the Administrative Task Scheduler (ATS). A job is created by adding a task to the ATS. The actual task name is constructed by concatenating the DBMS_JOB.TASK_NAME_PREFIX procedure name with the assigned job identifier, such as SAMPLE_JOB_TASK_1 where 1 is the job identifier.

A job runs a stored procedure which has been previously stored in the database. The SUBMIT procedure is used to create and store a job definition. A job identifier is assigned to every job, along with its associated stored procedure and the attributes describing when and how often the job is run.

On first run of the SUBMIT procedure in a database, the SYSTOOLSPACE table space is created if necessary.

To enable job scheduling for the DBMS_JOB routines, run:

```
db2set DB2_ATS_ENABLE=1
```

When and how often a job runs depends upon two interacting parameters – **next_date** and **interval**. The **next_date** parameter is a datetime value that specifies the next date and time when the job is to be executed. The **interval** parameter is a string that contains a date function that evaluates to a datetime value. Just prior to any execution of the job, the expression in the **interval** parameter is evaluated, and the resulting value replaces the **next_date** value stored with the job. The job is then executed. In this manner, the expression in **interval** is re-evaluated prior to each job execution, supplying the **next_date** date and time for the next execution.

The first run of a scheduled job, as specified by the **next_date** parameter, should be set at least 5 minutes after the current time, and the interval between running each job should also be at least 5 minutes.

The schema for this module is SYSIBMADM.

The DBMS_JOB module includes the following system-defined routines.

*Table 13. System-defined routines available in the DBMS_JOB module*

| Routine name | Description |
|---|---|
| BROKEN procedure | Specify that a given job is either broken or not broken. |
| CHANGE procedure | Change the parameters of the job. |
| INTERVAL procedure | Set the execution frequency by means of a date function that is recalculated each time the job runs. This value becomes the next date and time for execution. |
| NEXT_DATE procedure | Set the next date and time when the job is to be run. |
| REMOVE procedure | Delete the job definition from the database. |
| RUN procedure | Force execution of a job even if it is marked as broken. |
| SUBMIT procedure | Create a job and store the job definition in the database. |
| WHAT procedure | Change the stored procedure run by a job. |

*Table 14. System-defined constants available in the DBMS_JOB module*

| Constant name | Description |
|---|---|
| ANY_INSTANCE | The only supported value for the instance argument for the DBMS_JOB routines. |
| TASK_NAME_PREFIX | This constant contains the string that is used as the prefix for constructing the task name for the administrative task scheduler. |

## Usage notes

When the first job is submitted through the DBMS_JOB module for each database, the Administrative Task Scheduler setup is performed:

1. Create the SYSTOOLSPACE table space if it does not exist;
2. Create the ATS table and views, such as SYSTOOLS.ADMIN_TASK_LIST.

To list the scheduled jobs, run:

```
db2 SELECT * FROM systools.admin_task_list
      WHERE name LIKE DBMS_JOB.TASK_NAME_PREFIX || '_%'
```

To view the status of the job execution, run:

```
db2 SELECT * FROM systools.admin_task_status
      WHERE name LIKE DBMS_JOB.TASK_NAME_PREFIX || '_%'
```

## Examples

*Example 1:* The following example uses the stored procedure, `job_proc`. This stored procedure simply inserts a timestamp into the `jobrun` table, which contains a single VARCHAR column.

```
CREATE TABLE jobrun (
    runtime         VARCHAR(40)
)@
```

```
CREATE OR REPLACE PROCEDURE job_proc
BEGIN
    INSERT INTO jobrun VALUES ('job_proc run at ' || TO_CHAR(SYSDATE,
        'yyyy-mm-dd hh24:mi:ss'));
END@
```

This example results in the following output:

```
CREATE TABLE jobrun ( runtime        VARCHAR(40) )
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE PROCEDURE job_proc
BEGIN
    INSERT INTO jobrun VALUES ('job_proc run at ' || TO_CHAR(SYSDATE,
        'yyyy-mm-dd hh24:mi:ss'));
END
DB20000I  The SQL command completed successfully.
```

## BROKEN procedure - Set the state of a job to either broken or not broken

The BROKEN procedure sets the state of a job to either broken or not broken.

A broken job cannot be executed except by using the RUN procedure.

### Syntax

```
►►—BROKEN—(—job—,—broken——————————————)————————————————————————►◄
                        └,—next_date┘
```

### Parameters

*job* An input argument of type DECIMAL(20) that specifies the identifier of the job to be set as broken or not broken.

*broken*
An input argument of type BOOLEAN that specifies the job status. If set to "true", the job state is set to `broken`. If set to "false", the job state is set to `not broken`. Broken jobs cannot be run except through the RUN procedure.

*next_date*
An optional input argument of type DATE that specifies the date and time when the job runs. The default is `SYSDATE`.

### Authorization

EXECUTE privilege on the DBMS_JOB module.

### Examples

*Example 1:* Set the state of a job with job identifier 104 to broken:

```
CALL DBMS_JOB.BROKEN(104,true);
```

*Example 2:* Change the state back to not broken:

```
CALL DBMS_JOB.BROKEN(104,false);
```

## CHANGE procedure - Modify job attributes

The CHANGE procedure modifies certain job attributes, including the executable SQL statement, the next date and time the job is run, and how often it is run.

### Syntax

```
►►──CHANGE──(──job──,──what──,──next_date──,──interval──)──────────────────►◄
```

### Parameters

*job* An input argument of type DECIMAL(20) that specifies the identifier of the job with the attributes to modify.

*what*
An input argument of type VARCHAR(1024) that specifies the executable SQL statement. Set this argument to NULL if the existing value is to remain unchanged.

*next_date*
An input argument of type TIMESTAMP(0) that specifies the next date and time when the job is to run. Set this argument to NULL if the existing value is to remain unchanged.

*interval*
An input argument of type VARCHAR(1024) that specifies the date function that, when evaluated, provides the next date and time the job is to run. Set this argument to NULL if the existing value is to remain unchanged.

### Authorization

EXECUTE privilege on the DBMS_JOB module.

### Examples

*Example 1:* Change the job to run next on December 13, 2009. Leave other parameters unchanged.

```
CALL DBMS_JOB.CHANGE(104,NULL,TO_DATE('13-DEC-09','DD-MON-YY'),NULL);
```

## INTERVAL procedure - Set run frequency

The INTERVAL procedure sets the frequency of how often a job is run.

### Syntax

```
►►──INTERVAL──(──job──,──interval──)───────────────────────────────────────►◄
```

### Parameters

*job* An input argument of type DECIMAL(20) that specifies the identifier of the job whose frequency is being changed.

*interval*
An input argument of type VARCHAR(1024) that specifies the date function that, when evaluated, provides the next date and time the job is to run.

### Authorization

EXECUTE privilege on the DBMS_JOB module.

### Examples

*Example 1:* Change the job to run once a week:

```
CALL DBMS_JOB.INTERVAL(104,'SYSDATE + 7');
```

## NEXT_DATE procedure - Set the date and time when a job is run

The NEXT_DATE procedure sets the next date and time of when the job is to run.

### Syntax

```
►►—NEXT_DATE—(—job—,—next_date—)——————————————————————————————►◄
```

### Parameters

*job* An input argument of type DECIMAL(20) that specifies the identifier of the job whose next run date is to be modified.

*next_date*
An input argument of type TIMESTAMP(0) that specifies the date and time when the job is to be run next.

### Authorization

EXECUTE privilege on the DBMS_JOB module.

### Examples

*Example 1:* Change the job to run next on December 14, 2009:

```
CALL DBMS_JOB.NEXT_DATE(104, TO_DATE('14-DEC-09','DD-MON-YY'));
```

## REMOVE procedure - Delete the job definition from the database

The REMOVE procedure deletes the specified job from the database.

In order to have it executed again in the future, the job must be resubmitted using the SUBMIT procedure.

**Note:** The stored procedure associated with the job is not deleted when the job is removed.

### Syntax

```
►►—REMOVE—(—job—)—————————————————————————————————————————————►◄
```

### Parameters

*job* An input argument of type DECIMAL(20) that specifies the identifier of the job to be removed from the database.

### Authorization

EXECUTE privilege on the DBMS_JOB module.

### Examples

*Example 1:* Remove a job from the database:

```
CALL DBMS_JOB.REMOVE(104);
```

## RUN procedure - Force a broken job to run

The RUN procedure forces a job to run, even if it has a broken state.

### Syntax

```
►►──RUN──(──job──)────────────────────────────────────────────────────►◄
```

### Parameters

*job* An input argument of type DECIMAL(20) that specifies the identifier of the job to run.

### Authorization

EXECUTE privilege on the DBMS_JOB module.

### Examples

*Example 1:* Force a job to run.

```
CALL DBMS_JOB.RUN(104);
```

## SUBMIT procedure - Create a job definition and store it in the database

The SUBMIT procedure creates a job definition and stores it in the database.

A job consists of a job identifier, the stored procedure to be executed, when the job is first executed, and a date function that calculates the next date and time for the job to be run.

### Syntax

```
►►──SUBMIT──(──job──,──what──────────────────────────────────────────►

►─┬──────────────────────────────────────────┬─)──────────────────────►◄
  └─,──next_date──┬───────────────────────┬──┘
                  └─,──interval──┬──────────────┬─┘
                                 └─,──no_parse─┘
```

### Parameters

*job* An output argument of type DECIMAL(20) that specifies the identifier assigned to the job.

*what*
An input argument of type VARCHAR(1024) that specifies the name of the dynamically executable SQL statement.

*next_date*
An optional input argument of type TIMESTAMP(0) that specifies the next date and time when the job is to be run. The default is `SYSDATE`.

*interval*
An optional input argument of type VARCHAR(1024) that specifies a date function that, when evaluated, provides the date and time of the execution after the next execution. If *interval* is set to NULL, then the job is run only once. NULL is the default.

*no_parse*
An optional input argument of type BOOLEAN. If set to true, do not syntax-check the SQL statement at job creation; instead, perform syntax checking only when the job first executes. If set to false, syntax check the SQL statement at job creation. The default is false.

### Authorization

EXECUTE privilege on the DBMS_JOB module.

### Examples

*Example 1:* The following example creates a job using the stored procedure, `job_proc`. The job will first execute in about 5 minutes, and runs once a day thereafter as set by the *interval* argument, `SYSDATE + 1`.

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE jobid           INTEGER;
  CALL DBMS_JOB.SUBMIT(jobid,'CALL job_proc();',SYSDATE + 5 minutes, 'SYSDATE + 1');
  CALL DBMS_OUTPUT.PUT_LINE('jobid: ' || jobid);
END@
```

The output from this command is as follows:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

BEGIN
  DECLARE jobid           INTEGER;
  CALL DBMS_JOB.SUBMIT(jobid,'CALL job_proc();',SYSDATE + 5 minutes, 'SYSDATE + 1');
  CALL DBMS_OUTPUT.PUT_LINE('jobid: ' || jobid);
END
DB20000I  The SQL command completed successfully.

jobid: 1
```

## WHAT procedure - Change the SQL statement run by a job

The WHAT procedure changes the SQL statement run by a specified job.

### Syntax

```
►►—WHAT—(—job—,—what—)——————————————————————————————————————————►◄
```

### Parameters

*job* An input argument of type DECIMAL(20) that specifies the job identifier for which the dynamically executable SQL statement is to be changed.

*what*
An input argument of type VARCHAR(1024) that specifies the dynamically executed SQL statement.

### Authorization

EXECUTE privilege on the DBMS_JOB module.

### Examples

*Example 1:* Change the job to run the `list_emp` procedure:

```
CALL DBMS_JOB.WHAT(104,'list_emp;');
```

## DBMS_LOB module

The DBMS_LOB module provides the capability to operate on large objects.

In the following sections describing the individual procedures and functions, lengths and offsets are measured in bytes if the large objects are BLOBs. Lengths and offsets are measured in characters if the large objects are CLOBs.

The DBMS_LOB module supports LOB data up to 10M bytes.

The schema for this module is SYSIBMADM.

The DBMS_LOB module includes the following routines.

*Table 15. System-defined routines available in the DBMS_LOB module*

| Routine Name | Description |
|---|---|
| APPEND procedure | Appends one large object to another. |
| CLOSE procedure | Close an open large object. |
| COMPARE function | Compares two large objects. |
| CONVERTTOBLOB procedure | Converts character data to binary. |
| CONVERTTOCLOB procedure | Converts binary data to character. |
| COPY procedure | Copies one large object to another. |
| ERASE procedure | Erase a large object. |
| GET_STORAGE_LIMIT function | Get the storage limit for large objects. |
| GETLENGTH function | Get the length of the large object. |
| INSTR function | Get the position of the nth occurrence of a pattern in the large object starting at offset. |
| ISOPEN function | Check if the large object is open. |
| OPEN procedure | Open a large object. |
| READ procedure | Read a large object. |
| SUBSTR function | Get part of a large object. |
| TRIM procedure | Trim a large object to the specified length. |
| WRITE procedure | Write data to a large object. |
| WRITEAPPEND procedure | Write data from the buffer to the end of a large object. |

The following table lists the public variables available in the module.

*Table 16. DBMS_LOB public variables*

| Public variables | Data type | Value |
|---|---|---|
| lob_readonly | INTEGER | 0 |
| lob_readwrite | INTEGER | 1 |

## APPEND procedures - Append one large object to another

The APPEND procedures provide the capability to append one large object to another.

**Note:** Both large objects must be of the same type.

### Syntax

```
►►──APPEND_BLOB──(──dest_lob──,──src_lob──)─────────────────────────────►◄
```

```
►►──APPEND_CLOB──(──dest_lob──,──src_lob──)─────────────────────────────►◄
```

### Parameters

*dest_lob*
An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator for the destination object. Must be the same data type as *src_lob*.

*src_lob*
An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator for the source object. Must be the same data type as *dest_lob*.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## CLOSE procedures - Close an open large object

The CLOSE procedures are a no-op.

### Syntax

```
►►──CLOSE_BLOB──(──lob_loc──)───────────────────────────────────────────►◄
```

```
►►──CLOSE_CLOB──(──lob_loc──)───────────────────────────────────────────►◄
```

### Parameters

*lob_loc*
An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be closed.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

### Examples

*Example 1:*

## COMPARE function - Compare two large objects

The COMPARE function performs an exact byte-by-byte comparison of two large objects for a given length at given offsets.

The function returns:

- Zero if both large objects are exactly the same for the specified length for the specified offsets
- Non-zero if the objects are not the same
- Null if *amount*, *offset_1*, or *offset_2* are less than zero.

**Note:** The large objects being compared must be the same data type.

### Syntax

```
►►──COMPARE──(──lob_1──,──lob_2──────────────────────────────────────►

►──┬────────────────────────────────────────────┬──)────────────────►◄
   └─,──amount──┬────────────────────────────┬──┘
                └─,──offset_1──┬──────────────┬┘
                               └─,──offset_2──┘
```

### Parameters

*lob_1*
An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the first large object to be compared. Must be the same data type as *lob_2*.

*lob_2*
An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the second large object to be compared. Must be the same data type as *lob_1*.

*amount*
An optional input argument of type INTEGER. If the data type of the large objects is BLOB, then the comparison is made for *amount* bytes. If the data type of the large objects is CLOB, then the comparison is made for *amount* characters. The default is the maximum size of a large object.

*offset_1*
An optional input argument of type INTEGER that specifies the position within the first large object to begin the comparison. The first byte (or character) is offset 1. The default is 1.

*offset_2*
An optional input argument of type INTEGER that specifies the position within the second large object to begin the comparison. The first byte (or character) is offset 1. The default is 1.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## CONVERTTOBLOB procedure - Convert character data to binary

The CONVERTTOBLOB procedure provides the capability to convert character data to binary.

### Syntax

```
►►—CONVERTTOBLOB—(—dest_lob—,—src_clob—,—amount—,——————————————————————►

►—dest_offset—,—src_offset—,—blob_csid—,—lang_context—,—warning—)——————►◄
```

### Parameters

*dest_lob*
An input or output argument of type BLOB(10M) that specifies the large object locator into which the character data is to be converted.

*src_clob*
An input argument of type CLOB(10M) that specifies the large object locator of the character data to be converted.

*amount*
An input argument of type INTEGER that specifies the number of characters of *src_clob* to be converted.

*dest_offset*
An input or output argument of type INTEGER that specifies the position (in bytes) in the destination BLOB where writing of the source CLOB should begin. The first byte is offset 1.

*src_offset*
An input or output argument of type INTEGER that specifies the position (in characters) in the source CLOB where conversion to the destination BLOB should begin. The first character is offset 1.

*blob_csid*
An input argument of type INTEGER that specifies the character set ID of the destination BLOB. This value must match the database codepage.

*lang_context*
An input argument of type INTEGER that specifies the language context for the conversion. This value must be 0.

*warning*
An output argument of type INTEGER that always returns 0.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## CONVERTTOCLOB procedure - Convert binary data to character

The CONVERTTOCLOB procedure provides the capability to convert binary data to character.

### Syntax

```
►►──CONVERTTOCLOB──(──dest_lob──,──src_blob──,──amount──,──────────────────────►

►─dest_offset──,──src_offset──,──blob_csid──,──lang_context──,──warning──)────────►◄
```

### Parameters

*dest_lob*
: An input or output argument of type CLOB(10M) that specifies the large object locator into which the binary data is to be converted.

*src_clob*
: An input argument of type BLOB(10M) that specifies the large object locator of the binary data to be converted.

*amount*
: An input argument of type INTEGER that specifies the number of characters of *src_blob* to be converted.

*dest_offset*
: An input or output argument of type INTEGER that specifies the position (in characters) in the destination CLOB where writing of the source BLOB should begin. The first byte is offset 1.

*src_offset*
: An input or output argument of type INTEGER that specifies the position (in bytes) in the source BLOB where conversion to the destination CLOB should begin. The first character is offset 1.

*blob_csid*
: An input argument of type INTEGER that specifies the character set ID of the source BLOB. This value must match the database codepage.

*lang_context*
: An input argument of type INTEGER that specifies the language context for the conversion. This value must be 0.

*warning*
: An output argument of type INTEGER that always returns 0.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## COPY procedures - Copy one large object to another

The COPY procedures provide the capability to copy one large object to another.

**Note:** The source and destination large objects must be the same data type.

### Syntax

```
►►──COPY_BLOB──(──dest_lob──,──src_lob──,──amount──────────────────────────────────►

►──┬──────────────────────────────────────┬──)────────────────────────────────►◄
   └─,──dest_offset──┬─────────────────┬──┘
                     └─,──src_offset───┘
```

```
►►─COPY_CLOB─(─dest_lob─,─src_lob─,─amount──────────────────────────►

►─┬────────────────────────────────────┬─)──────────────────────────►◄
  └─,─dest_offset─┬──────────────────┬─┘
                  └─,─src_offset─────┘
```

### Parameters

*dest_lob*
An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to which *src_lob* is to be copied. Must be the same data type as *src_lob*.

*src_lob*
An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object from which *dest_lob* is to be copied. Must be the same data type as *dest_lob*.

*amount*
An input argument of type INTEGER that specifies the number of bytes or characters of *src_lob* to be copied.

*dest_offset*
An optional input argument of type INTEGER that specifies the position in the destination large object where writing of the source large object should begin. The first position is offset 1. The default is 1.

*src_offset*
An optional input argument of type INTEGER that specifies the position in the source large object where copying to the destination large object should begin. The first position is offset 1. The default is 1.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## ERASE procedures - Erase a portion of a large object

The ERASE procedures provide the capability to erase a portion of a large object.

To erase a large object means to replace the specified portion with zero-byte fillers for BLOBs or with spaces for CLOBs. The actual size of the large object is not altered.

### Syntax

```
►►─ERASE_BLOB─(─lob_loc─,─amount─┬─────────────┬─)──────────────────►◄
                                 └─,─offset────┘
```

```
►►─ERASE_CLOB─(─lob_loc─,─amount─┬─────────────┬─)──────────────────►◄
                                 └─,─offset────┘
```

### Parameters

*lob_loc*
An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be erased.

*amount*
: An input or output argument of type INTEGER that specifies the number of bytes or characters to be erased.

*offset*
: An optional input argument of type INTEGER that specifies the position in the large object where erasing is to begin. The first byte or character is at position 1. The default is 1.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## GET_STORAGE_LIMIT function - Return the limit on the largest allowable large object

The GET_STORAGE_LIMIT function returns the limit on the largest allowable large object.

The function returns an INTEGER value that reflects the maximum allowable size of a large object in this database.

### Syntax

```
►►──GET_STORAGE_LIMIT──(──)────────────────────────────────────────────►◄
```

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## GETLENGTH function - Return the length of the large object

The GETLENGTH function returns the length of a large object.

The function returns an INTEGER value that reflects the length of the large object in bytes (for a BLOB) or characters (for a CLOB).

### Syntax

```
►►──GETLENGTH──(──lob_loc──)───────────────────────────────────────────►◄
```

### Parameters

*lob_loc*
: An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object whose length is to be obtained.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## INSTR function - Return the location of the *n*th occurrence of a given pattern

The INSTR function returns the location of the *n*th occurrence of a given pattern within a large object.

The function returns an INTEGER value of the position within the large object where the pattern appears for the nth time, as specified by *nth*. This value starts from the position given by *offset*.

### Syntax

```
►►──INSTR──(──lob_loc──,──pattern──┬──────────────────────┬──)────────────►◄
                                   └─,──offset──┬────────┬┘
                                                └─,──nth─┘
```

### Parameters

*lob_loc*
An input argument of type BLOB or CLOB that specifies the large object locator of the large object in which to search for the *pattern*.

*pattern*
An input argument of type BLOB(32767) or VARCHAR(32672) that specifies the pattern of bytes or characters to match against the large object. Note that *pattern* must be BLOB if *lob_loc* is a BLOB; and *pattern* must be VARCHAR if *lob_loc* is a CLOB.

*offset*
An optional input argument of type INTEGER that specifies the position within *lob_loc* to start searching for the *pattern*. The first byte or character is at position 1. The default value is 1.

*nth*
An optional argument of type INTEGER that specifies the number of times to search for the *pattern*, starting at the position given by *offset*. The default value is 1.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## ISOPEN function - Test if the large object is open

The ISOPEN function always returns an INTEGER value of 1..

### Syntax

```
►►──ISOPEN──(──lob_loc──)──────────────────────────────────────────────►◄
```

### Parameters

*lob_loc*
An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be tested by the function.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## OPEN procedures - Open a large object

The OPEN procedures are a no-op.

### Syntax

```
►►──OPEN_BLOB──(──lob_loc──,──open_mode──)─────────────────────────────►◄
```

```
►►──OPEN_CLOB──(──lob_loc──,──open_mode──)─────────────────────────────►◄
```

### Parameters

*lob_loc*
: An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be opened.

*open_mode*
: An input argument of type INTEGER that specifies the mode in which to open the large object. Set to 0 (`lob_readonly`) for read-only mode. Set to 1 (`lob_readwrite`) for read-write mode.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## READ procedures - Read a portion of a large object

The READ procedures provide the capability to read a portion of a large object into a buffer.

### Syntax

```
►►──READ_BLOB──(──lob_loc──,──amount──,──offset──,──buffer──)──────────►◄
```

```
►►──READ_CLOB──(──lob_loc──,──amount──,──offset──,──buffer──)──────────►◄
```

### Parameters

*lob_loc*
: An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be read.

*amount*
: An input or output argument of type INTEGER that specifies the number of bytes or characters to read.

*offset*
: An input argument of type INTEGER that specifies the position to begin reading. The first byte or character is at position 1.

*buffer*
: An output argument of type BLOB(32762) or VARCHAR(32672) that specifies the variable to receive the large object. If *lob_loc* is a BLOB, then *buffer* must be BLOB. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## SUBSTR function - Return a portion of a large object

The SUBSTR function provides the capability to return a portion of a large object.

The function returns a BLOB(32767) (for a BLOB) or VARCHAR (for a CLOB) value for the returned portion of the large object read by the function.

### Syntax

```
►►──SUBSTR──(──lob_loc──┬─────────────────────────┬──)─────────────────────►◄
                        └─,──amount──┬───────────┬┘
                                     └─,──offset─┘
```

### Parameters

*lob_loc*
An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be read.

*amount*
An optional input argument of type INTEGER that specifies the number of bytes or characters to be returned. The default value is 32,767.

*offset*
An optional input argument of type INTEGER that specifies the position within the large object to begin returning data. The first byte or character is at position 1. The default value is 1.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## TRIM procedures - Truncate a large object to the specified length

The TRIM procedures provide the capability to truncate a large object to the specified length.

### Syntax

```
►►──TRIM_BLOB──(──lob_loc──,──newlen──)────────────────────────────────────►◄
```

```
►►──TRIM_CLOB──(──lob_loc──,──newlen──)────────────────────────────────────►◄
```

### Parameters

*lob_loc*
An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be trimmed.

*newlen*
An input argument of type INTEGER that specifies the new number of bytes or characters to which the large object is to be trimmed.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## WRITE procedures - Write data to a large object

The WRITE procedures provide the capability to write data into a large object.

Any existing data in the large object at the specified offset for the given length is overwritten by data given in the buffer.

### Syntax

```
►►──WRITE_BLOB──(──lob_loc──,──amount──,──offset──,──buffer──)──────────────────►◄
```

```
►►──WRITE_CLOB──(──lob_loc──,──amount──,──offset──,──buffer──)──────────────────►◄
```

### Parameters

*lob_loc*
An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be written.

*amount*
An input argument of type INTEGER that specifies the number of bytes or characters in *buffer* to be written to the large object.

*offset*
An input argument of type INTEGER that specifies the offset in bytes or characters from the beginning of the large object for the write operation to begin. The start value of the large object is 1.

*buffer*
An input argument of type BLOB(32767) or VARCHAR(32672) that contains the data to be written to the large object. If *lob_loc* is a BLOB, then *buffer* must be BLOB. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR.

### Authorization

EXECUTE privilege on the DBMS_LOB module.

## WRITEAPPEND procedures - Append data to the end of a large object

The WRITEAPPEND procedures provide the capability to add data to the end of a large object.

### Syntax

```
►►──WRITEAPPEND_BLOB──(──lob_loc──,──amount──,──buffer──)───────────────────────►◄
```

```
►►──WRITEAPPEND_CLOB──(──lob_loc──,──amount──,──buffer──)───────────────────────►◄
```

## Parameters

*lob_loc*

An input or output argument of type BLOB or CLOB that specifies the large object locator of the large object to which data is to appended.

*amount*

An input argument of type INTEGER that specifies the number of bytes or characters from *buffer* to be appended to the large object.

*buffer*

An input argument of type BLOB(32767) or VARCHAR(32672) that contains the data to be appended to the large object. If *lob_loc* is a BLOB, then *buffer* must be BLOB. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR.

## Authorization

EXECUTE privilege on the DBMS_LOB module.

# DBMS_OUTPUT module

The DBMS_OUTPUT module provides a set of procedures for putting messages (lines of text) in a message buffer and getting messages from the message buffer. These procedures are useful during application debugging when you need to write messages to standard output.

The schema for this module is SYSIBMADM.

The DBMS_OUTPUT module includes the following system-defined routines.

*Table 17. System-defined routines available in the DBMS_OUTPUT module*

| Routine name | Description |
|---|---|
| DISABLE procedure | Disables the message buffer. |
| ENABLE procedure | Enables the message buffer |
| GET_LINE procedure | Gets a line of text from the message buffer. |
| GET_LINES procedure | Gets one or more lines of text from the message buffer and places the text into a collection |
| NEW_LINE procedure | Puts an end-of-line character sequence in the message buffer. |
| PUT procedure | Puts a string that includes no end-of-line character sequence in the message buffer. |
| PUT_LINE procedure | Puts a single line that includes an end-of-line character sequence in the message buffer. |

The procedures in this module allow you to work with the message buffer. Use the command line processor (CLP) command SET SERVEROUTPUT ON to redirect the output to standard output.

## Example

In proc1 use the PUT and PUT_LINE procedures to put a line of text in the message buffer. When proc1 runs for the first time, SET SERVEROUTPUT ON is specified, and the line in the message buffer is printed to the CLP window. When proc1 runs a second time, SET SERVEROUTPUT OFF is specified, and no lines from the message buffer are printed to the CLP window.

```
CREATE PROCEDURE proc1( P1 VARCHAR(10) )
BEGIN
  CALL DBMS_OUTPUT.PUT( 'P1 = ' );
  CALL DBMS_OUTPUT.PUT_LINE( P1 );
END@

SET SERVEROUTPUT ON@

CALL proc1( '10' )@

SET SERVEROUTPUT OFF@

CALL proc1( '20' )@
```

The example results in the following output:

```
CALL proc1( '10' )

  Return Status = 0

P1 = 10

SET SERVEROUTPUT OFF
DB20000I  The SET SERVEROUTPUT command completed successfully.

CALL proc1( '20' )

  Return Status = 0
```

## DISABLE procedure - Disable the message buffer

The DISABLE procedure disables the message buffer.

After this procedure runs, any messages that are in the message buffer are discarded. Calls to the PUT, PUT_LINE, or NEW_LINE procedures are ignored, and no error is returned to the sender.

### Syntax

```
►►—DBMS_OUTPUT.DISABLE——————————————————————————————————————————————►◄
```

### Authorization

EXECUTE privilege on the DBMS_OUTPUT module.

### Example

The following example disables the message buffer for the current session:

```
CALL DBMS_OUTPUT.DISABLE@
```

### Usage notes

To send and receive messages after the message buffer has been disabled, use the ENABLE procedure.

## ENABLE procedure - Enable the message buffer

The ENABLE procedure enables the message buffer. During a single session, applications can put messages in the message buffer and get messages from the message buffer.

### Syntax

```
►►—DBMS_OUTPUT.ENABLE—(—buffer_size—)————————————————————————————————►◄
```

### Procedure parameters

*buffer_size*
: An input argument of type INTEGER that specifies the maximum length of the message buffer in bytes. If you specify a value of less than 2000 for *buffer_size*, the buffer size is set to 2000. If the value is NULL, then the default buffer size is 20000.

### Authorization

EXECUTE privilege on the DBMS_OUTPUT module.

### Example

The following example enables the message buffer:

```
CALL DBMS_OUTPUT.ENABLE( NULL )@
```

### Usage notes

You can call the ENABLE procedure to increase the size of an existing message buffer. Any messages in the old buffer are copied to the enlarged buffer.

## GET_LINE procedure - Get a line from the message buffer

The GET_LINE procedure gets a line of text from the message buffer. The text must be terminated by an end-of-line character sequence.

**Tip:** To add an end-of-line character sequence to the message buffer, use the PUT_LINE procedure, or, after a series of calls to the PUT procedure, use the NEW_LINE procedure.

### Syntax

```
►►—DBMS_OUTPUT.GET_LINE—(—line—,—status—)————————————————————————————►◄
```

### Procedure parameters

*line*
: An output argument of type VARCHAR(32672) that returns a line of text from the message buffer.

*status*
: An output argument of type INTEGER that indicates whether a line was returned from the message buffer:

- 0 indicates that a line was returned
- 1 indicates that there was no line to return

### Authorization

EXECUTE privilege on the DBMS_OUTPUT module.

### Example

Use the GET_LINE procedure to get a line of text from the message buffer. In this example, proc1 puts a line of text in the message buffer. proc3 gets the text from the message buffer and inserts it into a table named messages. proc2 then runs, but because the message buffer is disabled, no text is added to the message buffer. When the select statement runs, it returns only the text added by proc1.

```
CALL DBMS_OUTPUT.ENABLE( NULL )@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC1 put this line in the message buffer.' );
END@

CREATE PROCEDURE proc2()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC2 put this line in the message buffer.' );
END@

CREATE TABLE messages ( msg VARCHAR(100) )@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE line VARCHAR(32672);
  DECLARE status INT;

  CALL DBMS_OUTPUT.GET_LINE( line, status );
  while status = 0 do
    INSERT INTO messages VALUES ( line );
    CALL DBMS_OUTPUT.GET_LINE( line, status );
  end while;
END@

CALL proc1@

CALL proc3@

CALL DBMS_OUTPUT.DISABLE@

CALL proc2@

CALL proc3@

SELECT * FROM messages@
```

This example results in the following output:

```
MSG
----------------------------------------------
PROC1 put this line in the message buffer.

  1 record(s) selected.
```

## GET_LINES procedure - Get multiple lines from the message buffer

The GET_LINES procedure gets one or more lines of text from the message buffer and stores the text in a collection. Each line of text must be terminated by an end-of-line character sequence.

**Tip:** To add an end-of-line character sequence to the message buffer, use the PUT_LINE procedure, or, after a series of calls to the PUT procedure, use the NEW_LINE procedure.

## Syntax

```
►►—DBMS_OUTPUT.GET_LINES—(—lines—,—numlines—)———————————————————————————————►◄
```

## Procedure parameters

*lines*
An output argument of type DBMS_OUTPUT.CHARARR that returns the lines of text from the message buffer. The type DBMS_OUTPUT.CHARARR is internally defined as a VARCHAR(32672) ARRAY[2147483647] array.

*numlines*
An input and output argument of type INTEGER. When used as input, specifies the number of lines to retrieve from the message buffer. When used as output, indicates the actual number of lines that were retrieved from the message buffer. If the output value of *numlines* is less than the input value, then there are no more lines remaining in the message buffer.

## Authorization

EXECUTE privilege on the DBMS_OUTPUT module.

## Example

Use the GET_LINES procedure to get lines of text from the message buffer and store the text in an array. The text in the array can be inserted into a table and queried.

```
CALL DBMS_OUTPUT.ENABLE( NULL )@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC1 put this line in the message buffer.' );
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC1 put this line in the message buffer.' );
END@

CREATE PROCEDURE proc2()
BEGIN
  CALL DBMS_OUTPUT.PUT_LINE( 'PROC2 put this line in the message buffer.' );
END@

CREATE TABLE messages ( msg VARCHAR(100) )@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE lines DBMS_OUTPUT.CHARARR;
  DECLARE numlines INT;
  DECLARE i INT;

  CALL DBMS_OUTPUT.GET_LINES( lines, numlines );
  SET i = 1;
  WHILE i <= numlines DO
    INSERT INTO messages VALUES ( lines[i] );
    SET i = i + 1;
  END WHILE;
END@

CALL proc1@
CALL proc3@

CALL DBMS_OUTPUT.DISABLE@

CALL proc2@

CALL proc3@

SELECT * FROM messages@
```

This example results in the following output:

```
MSG
------------------------------------------
PROC1 put this line in the message buffer.
PROC1 put this line in the message buffer

  2 record(s) selected.
```

## NEW_LINE procedure - Put an end-of-line character sequence in the message buffer

The NEW_LINE procedure puts an end-of-line character sequence in the message buffer.

### Syntax

```
►►──DBMS_OUTPUT.NEW_LINE──────────────────────────────────────────────────►◄
```

### Authorization

EXECUTE privilege on the DBMS_OUTPUT module.

### Example

Use the NEW_LINE procedure to write an end-of-line character sequence to the message buffer. In this example, the text that is followed by an end-of-line character sequence displays as output because SET SERVEROUTPUT ON is specified. However, the text that is in the message buffer, but is not followed by an end-of-line character, does not display.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT( 'T' );
  CALL DBMS_OUTPUT.PUT( 'h' );
  CALL DBMS_OUTPUT.PUT( 'i' );
  CALL DBMS_OUTPUT.PUT( 's' );
  CALL DBMS_OUTPUT.NEW_LINE;
  CALL DBMS_OUTPUT.PUT( 'T' );
  CALL DBMS_OUTPUT.PUT( 'h' );
  CALL DBMS_OUTPUT.PUT( 'a' );
  CALL DBMS_OUTPUT.PUT( 't' );
END@

CALL proc1@

SET SERVEROUTPUT OFF@
```

This example results in the following output:

```
This
```

## PUT procedure - Put a partial line in the message buffer

The PUT procedure puts a string in the message buffer. No end-of-line character sequence is written at the end of the string.

### Syntax

```
►►—DBMS_OUTPUT.PUT—(—item—)————————————————————————————————————————►◄
```

### Procedure parameters

*item*
An input argument of type VARCHAR(32672) that specifies the text to write to the message buffer.

### Authorization

EXECUTE privilege on the DBMS_OUTPUT module.

### Example

Use the PUT procedure to put a partial line in the message buffer. In this example, the NEW_LINE procedure adds an end-of-line character sequence to the message buffer. When proc1 runs, because SET SERVEROUTPUT ON is specified, a line of text is returned.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  CALL DBMS_OUTPUT.PUT( 'H' );
  CALL DBMS_OUTPUT.PUT( 'e' );
  CALL DBMS_OUTPUT.PUT( 'l' );
  CALL DBMS_OUTPUT.PUT( 'l' );
  CALL DBMS_OUTPUT.PUT( 'o' );
  CALL DBMS_OUTPUT.PUT( '.' );
  CALL DBMS_OUTPUT.NEW_LINE;
END@

CALL proc1@

SET SERVEROUTPUT OFF@
```

This example results in the following output:

```
Hello.
```

### Usage notes

After using the PUT procedure to add text to the message buffer, use the NEW_LINE procedure to add an end-of-line character sequence to the message buffer. Otherwise, the text is not returned by the GET_LINE and GET_LINES procedures because it is not a complete line.

## PUT_LINE procedure - Put a complete line in the message buffer

The PUT_LINE procedure puts a single line that includes an end-of-line character sequence in the message buffer.

### Syntax

```
►►──DBMS_OUTPUT.PUT_LINE──(──item──)──────────────────────────────────────►◄
```

### Procedure parameters

*item*
An input argument of type VARCHAR(32672) that specifies the text to write to the message buffer.

### Authorization

EXECUTE privilege on the PUT_LINE procedure.

### Example

Use the PUT_LINE procedure to write a line that includes an end-of-line character sequence to the message buffer.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE PROC1()
BEGIN
  CALL DBMS_OUTPUT.PUT( 'a' );
  CALL DBMS_OUTPUT.NEW_LINE;
  CALL DBMS_OUTPUT.PUT_LINE( 'b' );
END@

CALL PROC1@

SET SERVEROUTPUT OFF@
```

This example results in the following output:

```
a
b
```

# DBMS_PIPE module

The DBMS_PIPE module provides a set of routines for sending messages through a pipe within or between sessions that are connected to the same database.

The schema for this module is SYSIBMADM.

The DBMS_PIPE module includes the following system-defined routines.

*Table 18. System-defined routines available in the DBMS_PIPE module*

| Routine name | Description |
|---|---|
| CREATE_PIPE function | Explicitly creates a private or public pipe. |
| NEXT_ITEM_TYPE function | Determines the data type of the next item in a received message. |

*Table 18. System-defined routines available in the DBMS_PIPE module (continued)*

| Routine name | Description |
|---|---|
| PACK_MESSAGE function | Puts an item in the session’s local message buffer. |
| PACK_MESSAGE_RAW procedure | Puts an item of type RAW in the session's local message buffer. |
| PURGE procedure | Removes unreceived messages in the specified pipe. |
| RECEIVE_MESSAGE function | Gets a message from the specified pipe. |
| REMOVE_PIPE function | Deletes an explicitly created pipe. |
| RESET_BUFFER procedure | Resets the local message buffer. |
| SEND_MESSAGE procedure | Sends a message on the specified pipe. |
| UNIQUE_SESSION_NAME function | Returns a unique session name. |
| UNPACK_MESSAGE procedures | Retrieves the next data item from a message and assigns it to a variable. |

## Usage notes

Pipes are created either implicitly or explicitly during procedure calls. An *implicit pipe* is created when a procedure call contains a reference to a pipe name that does not exist. For example, if a pipe named "mailbox" is passed to the SEND_MESSAGE procedure and that pipe does not already exist, a new pipe named "mailbox" is created. An *explicit pipe* is created by calling the CREATE_PIPE function and specifying the name of the pipe.

Pipes can be private or public. A *private pipe* can only be accessed by the user who created the pipe. Even an administrator cannot access a private pipe that was created by another user. A *public pipe* can be accessed by any user who has access to the DBMS_PIPE module. To specify the access level for a pipe, use the CREATE_PIPE function and specify a value for the *private* parameter: "false" specifies that the pipe is public; "true" specifies that the pipe is private. If no value is specified, the default is to create a private pipe. All implicit pipes are private.

To send a message through a pipe, call the PACK_MESSAGE function to put individual data items (lines) in a local message buffer that is unique to the current session. Then, call the SEND_MESSAGE procedure to send the message through the pipe.

To receive a message, call the RECEIVE_MESSAGE function to get a message from the specified pipe. The message is written to the receiving session’s local message buffer. Then, call the UNPACK_MESSAGE procedure to retrieve the next data item from the local message buffer and assign it to a specified program variable. If a pipe contains multiple messages, the RECEIVE_MESSAGE function gets the messages in FIFO (first-in-first-out) order.

Each session maintains separate message buffers for messages that are created by the PACK_MESSAGE function and messages that are retrieved by the RECEIVE_MESSAGE function. The separate message buffers allow you to build and receive messages in the same session. However, when consecutive calls are made to the RECEIVE_MESSAGE function, only the message from the last RECEIVE_MESSAGE call is preserved in the local message buffer.

### Example

In connection 1, create a pipe that is named pipe1. Put a message in the session's local message buffer, and send the message through pipe1.

```
BEGIN
  DECLARE status INT;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@
```

In connection 2, receive the message, unpack it, and display it to standard output.

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE status   INT;
  DECLARE int1     INTEGER;
  DECLARE date1    DATE;
  DECLARE raw1     BLOB(100);
  DECLARE varchar1 VARCHAR(100);
  DECLARE itemType INTEGER;

  SET status = DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
  IF( status = 0 ) THEN
    SET itemType = DBMS_PIPE.NEXT_ITEM_TYPE();
    CASE itemType
      WHEN 6 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_INT( int1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'int1: ' || int1 );
      WHEN 9 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR( varchar1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'varchar1: ' || varchar1 );
      WHEN 12 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_DATE( date1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'date1:' || date1 );
      WHEN 23 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_RAW( raw1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'raw1: ' || VARCHAR(raw1) );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'Unexpected value' );
    END CASE;
  END IF;
  SET status = DBMS_PIPE.REMOVE_PIPE( 'pipe1' );
END@
```

This example results in the following output:

```
  varchar1: message1
```

## CREATE_PIPE function - Create a pipe

The CREATE_PIPE function explicitly creates a public or private pipe with the specified name.

For more information about explicit public and private pipes, see the topic about the DBMS_PIPE module.

### Syntax

```
►►──DBMS_PIPE.CREATE_PIPE──(──pipename──,──┬─────────────┬──,──┬─────────┬──)──────►◄
                                           └─maxpipesize─┘     └─private─┘
```

## Return value

This function returns the status code 0 if the pipe is created successfully.

## Function parameters

*pipename*
An input argument of type VARCHAR(128) that specifies the name of the pipe. For more information about pipes, see “DBMS_PIPE module” on page 233.

*maxpipesize*
An optional input argument of type INTEGER that specifies the maximum capacity of the pipe in bytes. The default is 8192 bytes.

*private*
An optional input argument that specifies the access level of the pipe:

**For non-partitioned database environments**
A value of "0" or "FALSE" creates a public pipe.

A value of "1" or "TRUE creates a private pipe. This is the default.

**In a partitioned database environment**
A value of "0" creates a public pipe.

A value of "1" creates a private pipe. This is the default.

## Authorization

EXECUTE privilege on the DBMS_PIPE module.

## Example

*Example 1:* Create a private pipe that is named messages:

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_status        INTEGER;
  SET v_status = DBMS_PIPE.CREATE_PIPE('messages');
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END@

CALL proc1@
```

This example results in the following output:

```
CREATE_PIPE status: 0
```

*Example 2:* Create a public pipe that is named mailbox:

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE v_status INTEGER;
  SET v_status = DBMS_PIPE.CREATE_PIPE('mailbox',0);
  DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status: ' || v_status);
END@

CALL proc2@
```

This example results in the following output:

```
CREATE_PIPE status: 0
```

## NEXT_ITEM_TYPE function - Return the data type code of the next item

The NEXT_ITEM_TYPE function returns an integer code that identifies the data type of the next data item in a received message.

The received message is stored in the session's local message buffer. Use the UNPACK_MESSAGE procedure to move each item off of the local message buffer, and then use the NEXT_ITEM_TYPE function to return the data type code for the next available item. A code of 0 is returned when there are no more items left in the message.

### Syntax

```
►►──DBMS_PIPE.NEXT_ITEM_TYPE──────────────────────────────────────────────►◄
```

### Return value

This function returns one of the following codes that represents a data type.

*Table 19. NEXT_ITEM_TYPE data type codes*

| Type code | Data type |
|---|---|
| 0 | No more data items |
| 6 | INTEGER |
| 9 | VARCHAR |
| 12 | DATE |
| 23 | BLOB |

### Authorization

EXECUTE privilege on the DBMS_PIPE module.

### Example

In proc1, pack and send a message. In proc2, receive the message and then unpack it by using the NEXT_ITEM_TYPE function to determine its type.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INT;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE status   INT;
  DECLARE num1     DECFLOAT;
  DECLARE date1    DATE;
  DECLARE raw1     BLOB(100);
  DECLARE varchar1 VARCHAR(100);
  DECLARE itemType INTEGER;

  SET status = DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
  IF( status = 0 ) THEN
    SET itemType = DBMS_PIPE.NEXT_ITEM_TYPE();
    CASE itemType
      WHEN 6 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_NUMBER( num1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'num1: ' || num1 );
      WHEN 9 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR( varchar1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'varchar1: ' || varchar1 );
      WHEN 12 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_DATE( date1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'date1:' || date1 );
      WHEN 23 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_RAW( raw1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'raw1: ' || VARCHAR(raw1) );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'Unexpected value' );
    END CASE;
  END IF;
  SET status = DBMS_PIPE.REMOVE_PIPE( 'pipe1' );
END@

CALL proc1@

CALL proc2@
```

This example results in the following output:

```
varchar1: message1
```

## PACK_MESSAGE function - Put a data item in the local message buffer

The PACK_MESSAGE function puts a data item in the session's local message buffer.

### Syntax

```
►►—DBMS_PIPE.PACK_MESSAGE—(—item—)————————————————————————————————►◄
```

### Procedure parameters

*item*

An input argument of type VARCHAR(4096), DATE, or DECFLOAT that contains an expression. The value returned by this expression is added to the local message buffer of the session.

**Tip:** To put data items of type RAW in the local message buffer, use the PACK_MESSAGE_RAW procedure.

### Authorization

EXECUTE privilege on the DBMS_PIPE module.

### Example

Use the PACK_MESSAGE function to put a message for Sujata in the local message buffer, and then use the SEND_MESSAGE procedure to send the message on a pipe.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_status    INTEGER;
  DECLARE    status      INTEGER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Hi, Sujata');
  SET status = DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?');
  SET status = DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CALL proc1@
```

This example results in the following output:

```
  SEND_MESSAGE status: 0
```

### Usage notes

The PACK_MESSAGE function or PACK_MESSAGE_RAW procedure must be called at least once before issuing a SEND_MESSAGE call.

## PACK_MESSAGE_RAW procedure - Put a data item of type RAW in the local message buffer

The PACK_MESSAGE_RAW procedure puts a data item of type RAW in the session's local message buffer.

### Syntax

```
►►──DBMS_PIPE.PACK_MESSAGE_RAW──(──item──)──────────────────────────────►◄
```

### Procedure parameters

*item*

An input argument of type BLOB(4096) that specifies an expression. The value returned by this expression is added to the session's local message buffer.

### Authorization

EXECUTE privilege on the DBMS_PIPE module.

### Example

Use the PACK_MESSAGE_RAW procedure to put a data item of type RAW in the local message buffer.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_raw           BLOB(100);
  DECLARE v_raw2   BLOB(100);
  DECLARE v_status        INTEGER;
  SET v_raw = BLOB('21222324');
  SET v_raw2 = BLOB('30000392');
  CALL DBMS_PIPE.PACK_MESSAGE_RAW(v_raw);
  CALL DBMS_PIPE.PACK_MESSAGE_RAW(v_raw2);
  SET v_status = DBMS_PIPE.SEND_MESSAGE('datatypes');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CALL proc1@
```

This example results in the following output:

```
SEND_MESSAGE status: 0
```

### Usage notes

The PACK_MESSAGE function or PACK_MESSAGE_RAW procedure must be called at least once before issuing a SEND_MESSAGE call.

## PURGE procedure - Remove unreceived messages from a pipe

The PURGE procedure removes unreceived messages in the specified implicit pipe.

**Tip:** Use the REMOVE_PIPE function to delete an explicit pipe.

### Syntax

```
►►—DBMS_PIPE.PURGE—(—pipename—)——————————————————————————————————►◄
```

### Procedure parameters

*pipename*
An input argument of type VARCHAR(128) that specifies the name of the implicit pipe.

### Authorization

EXECUTE privilege on the DBMS_PIPE module.

### Example

In proc1 send two messages on a pipe: Message #1 and Message #2. In proc2, receive the first message, unpack it, and then purge the pipe. When proc3 runs, the call to the RECEIVE_MESSAGE function times out and returns the status code 1 because no message is available.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE   v_status        INTEGER;
  DECLARE   status          INTEGER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Message #1');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
  SET status = DBMS_PIPE.PACK_MESSAGE('Message #2');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE   v_item          VARCHAR(80);
  DECLARE   v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  CALL DBMS_PIPE.PURGE('pipe');
END@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END@

CALL proc1@

CALL proc2@

CALL proc3@
```

This example results in the following output.

From proc1:

```
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

From proc2:

```
RECEIVE_MESSAGE status: 0
Item: Hi, Sujata
```

From proc3:

```
RECEIVE_MESSAGE status: 1
```

## RECEIVE_MESSAGE function - Get a message from a specified pipe

The RECEIVE_MESSAGE function gets a message from a specified pipe.

### Syntax

```
►►──DBMS_PIPE.RECEIVE_MESSAGE──(──pipename──┬──────────────┬──)─────────────────►◄
                                            └─,──timeout───┘
```

### Return value

The RECEIVE_MESSAGE function returns one of the following status codes of type INTEGER.

*Table 20. RECEIVE_MESSAGE status codes*

| Status code | Description |
|---|---|
| 0 | Success |
| 1 | Time out |

### Function parameters

*pipename*
An input argument of type VARCHAR(128) that specifies the name of the pipe.

If the specified pipe does not exist, the pipe is created implicitly. For more information about pipes, see “DBMS_PIPE module” on page 233.

*timeout*
An optional input argument of type INTEGER that specifies the wait time in seconds. The default is 86400000 (1000 days).

### Authorization

EXECUTE privilege on the DBMS_PIPE module.

### Example

In proc1, send a message. In proc2, receive and unpack the message. Timeout if the message is not received within 1 second.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INTEGER;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END@

CALL proc1@
CALL proc2@
```

This example results in the following output:

```
RECEIVE_MESSAGE status: 0
Item: message1
```

## REMOVE_PIPE function - Delete a pipe

The REMOVE_PIPE function deletes an explicitly created pipe. Use this function to delete any public or private pipe that was created by the CREATE_PIPE function.

### Syntax

```
►►─DBMS_PIPE.REMOVE_PIPE─(─pipename─)───────────────────────────►◄
```

### Return value

This function returns one of the following status codes of type INTEGER.

*Table 21. REMOVE_PIPE status codes*

| Status code | Description |
|---|---|
| 0 | Pipe successfully removed or does not exist |

*Table 21. REMOVE_PIPE status codes (continued)*

| Status code | Description |
|---|---|
| NULL | An exception is thrown |

## Function parameters

*pipename*
An input argument of type VARCHAR(128) that specifies the name of the pipe.

## Authorization

EXECUTE privilege on the DBMS_PIPE module.

## Example

In proc1 send two messages on a pipe: Message #1 and Message #2. In proc2, receive the first message, unpack it, and then delete the pipe. When proc3 runs, the call to the RECEIVE_MESSAGE function times out and returns the status code 1 because the pipe no longer exists.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_status        INTEGER;
  DECLARE    status          INTEGER;
  SET  v_status = DBMS_PIPE.CREATE_PIPE('pipe1');
  CALL DBMS_OUTPUT.PUT_LINE('CREATE_PIPE status : ' || v_status);

  SET status = DBMS_PIPE.PACK_MESSAGE('Message #1');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe1');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);

  SET status = DBMS_PIPE.PACK_MESSAGE('Message #2');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe1');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  DECLARE    status          INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  SET status = DBMS_PIPE.REMOVE_PIPE('pipe1');
END@

CREATE PROCEDURE proc3()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
END@

CALL proc1@

CALL proc2@

CALL proc3@
```

This example results in the following output.

From proc1:

```
CREATE_PIPE status : 0
SEND_MESSAGE status: 0
SEND_MESSAGE status: 0
```

From proc2:

```
RECEIVE_MESSAGE status: 0
Item: Message #1
```

From proc3:

```
RECEIVE_MESSAGE status: 1
```

## RESET_BUFFER procedure - Reset the local message buffer

The RESET_BUFFER procedure resets a pointer to the session's local message buffer back to the beginning of the buffer. Resetting the buffer causes subsequent PACK_MESSAGE calls to overwrite any data items that existed in the message buffer prior to the RESET_BUFFER call.

### Syntax

```
►►──DBMS_PIPE.RESET_BUFFER──────────────────────────────────────────────►◄
```

### Authorization

EXECUTE privilege on the DBMS_PIPE module.

### Example

In proc1, use the PACK_MESSAGE function to put a message for an employee named Sujata in the local message buffer. Call the RESET_BUFFER procedure to replace the message with a message for Bing, and then send the message on a pipe. In proc2, receive and unpack the message for Bing.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_status        INTEGER;
  DECLARE    status          INTEGER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Hi, Sujata');
  SET status = DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 3:00, today?');
  SET status = DBMS_PIPE.PACK_MESSAGE('If not, is tomorrow at 8:30 ok with you?');
  CALL DBMS_PIPE.RESET_BUFFER;
  SET status = DBMS_PIPE.PACK_MESSAGE('Hi, Bing');
  SET status = DBMS_PIPE.PACK_MESSAGE('Can you attend a meeting at 9:30, tomorrow?');
  SET v_status = DBMS_PIPE.SEND_MESSAGE('pipe');
  CALL DBMS_OUTPUT.PUT_LINE('SEND_MESSAGE status: ' || v_status);
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
```

```
END@

CALL proc1@

CALL proc2@
```

This example results in the following output:

From proc1:

```
SEND_MESSAGE status: 0
```

From proc2:

```
RECEIVE_MESSAGE status: 0
Item: Hi, Bing
Item: Can you attend a meeting at 9:30, tomorrow?
```

## SEND_MESSAGE procedure - Send a message to a specified pipe

The SEND_MESSAGE procedure sends a message from the session's local message buffer to a specified pipe.

### Syntax

```
►►──DBMS_PIPE.SEND_MESSAGE──(──pipename──┬──────────────┬──┬──────────────────┬──)──────────►◄
                                          └─,──timeout──┘  └─,──maxpipesize──┘
```

### Return value

This procedure returns one of the following status codes of type INTEGER.

*Table 22. SEND_MESSAGE status codes*

| Status code | Description |
|---|---|
| 0 | Success |
| 1 | Time out |

### Procedure parameters

*pipename*
: An input argument of type VARCHAR(128) that specifies the name of the pipe. If the specified pipe does not exist, the pipe is created implicitly. For more information about pipes, see "DBMS_PIPE module" on page 233.

*timeout*
: An optional input argument of type INTEGER that specifies the wait time in seconds. The default is 86400000 (1000 days).

*maxpipesize*
: An optional input argument of type INTEGER that specifies the maximum capacity of the pipe in bytes. The default is 8192 bytes.

### Authorization

EXECUTE privilege on the DBMS_PIPE module.

### Example

In proc1, send a message. In proc2, receive and unpack the message. Timeout if the message is not received within 1 second.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INTEGER;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE    v_item          VARCHAR(80);
  DECLARE    v_status        INTEGER;
  SET v_status = DBMS_PIPE.RECEIVE_MESSAGE('pipe1',1);
  CALL DBMS_OUTPUT.PUT_LINE('RECEIVE_MESSAGE status: ' || v_status);
  CALL DBMS_PIPE.UNPACK_MESSAGE(v_item);
  CALL DBMS_OUTPUT.PUT_LINE('Item: ' || v_item);
END@

CALL proc1@
CALL proc2@
```

This example results in the following output:

```
RECEIVE_MESSAGE status: 0
Item: message1
```

## UNIQUE_SESSION_NAME function - Return a unique session name

The UNIQUE_SESSION_NAME function returns a unique name for the current session.

You can use this function to create a pipe that has the same name as the current session. To create this pipe, pass the value returned by the UNIQUE_SESSION_NAME function to the SEND_MESSAGE procedure as the pipe name. An implicit pipe is created that has the same name as the current session.

### Syntax

```
►►──DBMS_PIPE.UNIQUE_SESSION_NAME──────────────────────────────────────────►◄
```

### Return value

This function returns a value of type VARCHAR(128) that represents the unique name for the current session.

### Authorization

EXECUTE privilege on the DBMS_PIPE module.

### Example

Create a pipe that has the same name as the current session.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    status          INTEGER;
  DECLARE    v_session       VARCHAR(30);
  SET v_session = DBMS_PIPE.UNIQUE_SESSION_NAME;
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE(v_session);
  CALL DBMS_OUTPUT.PUT_LINE('Sent message on pipe ' || v_session);
END@

CALL proc1@
```

This example results in the following output:

```
Sent message on pipe *LOCAL.myschema.080522010048
```

## UNPACK_MESSAGE procedures - Get a data item from the local message buffer

The UNPACK_MESSAGE procedures retrieve the next data item from a message and assign it to a variable.

Before calling one of the UNPACK_MESSAGE procedures, use the RECEIVE_MESSAGE procedure to place the message in the local message buffer.

### Syntax

```
►►──DBMS_PIPE.UNPACK_MESSAGE_NUMBER──(──item──)──────────────────────────►◄
```

```
►►──DBMS_PIPE.UNPACK_MESSAGE_CHAR──(──item──)────────────────────────────►◄
```

```
►►──DBMS_PIPE.UNPACK_MESSAGE_DATE──(──item──)────────────────────────────►◄
```

```
►►──DBMS_PIPE.UNPACK_MESSAGE_RAW──(──item──)─────────────────────────────►◄
```

### Procedure parameters

*item*

An output argument of one of the following types that specifies a variable to receive data items from the local message buffer.

| Routine | Data type |
|---|---|
| UNPACK_MESSAGE_NUMBER | DECFLOAT |
| UNPACK_MESSAGE_CHAR | VARCHAR(4096) |
| UNPACK_MESSAGE_DATE | DATE |
| UNPACK_MESSAGE_RAW | BLOB(4096) |

## Authorization

EXECUTE privilege on the DBMS_PIPE module.

## Example

In proc1, pack and send a message. In proc2, receive the message, unpack it using the appropriate procedure based on the item's type, and display the message to standard output.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE status INT;
  SET status = DBMS_PIPE.CREATE_PIPE( 'pipe1' );
  SET status = DBMS_PIPE.PACK_MESSAGE('message1');
  SET status = DBMS_PIPE.SEND_MESSAGE( 'pipe1' );
END@

CREATE PROCEDURE proc2()
BEGIN
  DECLARE status   INT;
  DECLARE num1     DECFLOAT;
  DECLARE date1    DATE;
  DECLARE raw1     BLOB(100);
  DECLARE varchar1 VARCHAR(100);
  DECLARE itemType INTEGER;

  SET status = DBMS_PIPE.RECEIVE_MESSAGE( 'pipe1' );
  IF( status = 0 ) THEN
    SET itemType = DBMS_PIPE.NEXT_ITEM_TYPE();
    CASE itemType
      WHEN 6 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_NUMBER( num1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'num1: ' || num1 );
      WHEN 9 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_CHAR( varchar1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'varchar1: ' || varchar1 );
      WHEN 12 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_DATE( date1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'date1:' || date1 );
      WHEN 23 THEN
        CALL DBMS_PIPE.UNPACK_MESSAGE_RAW( raw1 );
        CALL DBMS_OUTPUT.PUT_LINE( 'raw1: ' || VARCHAR(raw1) );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'Unexpected value' );
    END CASE;
  END IF;
  SET status = DBMS_PIPE.REMOVE_PIPE( 'pipe1' );
END@

CALL proc1@

CALL proc2@
```

This example results in the following output:

```
varchar1: message1
```

# DBMS_SQL module

The DBMS_SQL module provides a set of procedures for executing dynamic SQL, and therefore supports various data manipulation language (DML) or data definition language (DDL) statement.

The schema for this module is SYSIBMADM.

The DBMS_SQL module includes the following system-defined routines.

*Table 23. System-defined routines available in the DBMS_SQL module*

| Procedure name | Description |
|---|---|
| BIND_VARIABLE_BLOB procedure | Provides the input BLOB value for the IN or INOUT parameter; and defines the data type of the output value to be BLOB for the INOUT or OUT parameter. |
| BIND_VARIABLE_CHAR procedure | Provides the input CHAR value for the IN or INOUT parameter; and defines the data type of the output value to be CHAR for the INOUT or OUT parameter. |
| BIND_VARIABLE_CLOB procedure | Provides the input CLOB value for the IN or INOUT parameter; and defines the data type of the output value to be CLOB for the INOUT or OUT parameter. |
| BIND_VARIABLE_DATE procedure | Provides the input DATE value for the IN or INOUT parameter; and defines the data type of the output value to be DATE for the INOUT or OUT parameter. |
| BIND_VARIABLE_DOUBLE procedure | Provides the input DOUBLE value for the IN or INOUT parameter; and defines the data type of the output value to be DOUBLE for the INOUT or OUT parameter. |
| BIND_VARIABLE_INT procedure | Provides the input INTEGER value for the IN or INOUT parameter; and defines the data type of the output value to be INTEGER for the INOUT or OUT parameter. |
| BIND_VARIABLE_NUMBER procedure | Provides the input DECFLOAT value for the IN or INOUT parameter; and defines the data type of the output value to be DECFLOAT for the INOUT or OUT parameter. |
| BIND_VARIABLE_RAW procedure | Provides the input BLOB(32767) value for the IN or INOUT parameter; and defines the data type of the output value to be BLOB(32767) for the INOUT or OUT parameter. |
| BIND_VARIABLE_TIMESTAMP procedure | Provides the input TIMESTAMP value for the IN or INOUT parameter; and defines the data type of the output value to be TIMESTAMP for the INOUT or OUT parameter. |

*Table 23. System-defined routines available in the DBMS_SQL module (continued)*

| Procedure name | Description |
|---|---|
| BIND_VARIABLE_VARCHAR procedure | Provides the input VARCHAR value for the IN or INOUT parameter; and defines the data type of the output value to be VARCHAR for the INOUT or OUT parameter. |
| CLOSE_CURSOR procedure | Closes a cursor. |
| COLUMN_VALUE_BLOB procedure | Retrieves the value of column of type BLOB. |
| COLUMN_VALUE_CHAR procedure | Retrieves the value of column of type CHAR. |
| COLUMN_VALUE_CLOB procedure | Retrieves the value of column of type CLOB. |
| COLUMN_VALUE_DATE procedure | Retrieves the value of column of type DATE. |
| COLUMN_VALUE_DOUBLE procedure | Retrieves the value of column of type DOUBLE. |
| COLUMN_VALUE_INT procedure | Retrieves the value of column of type INTEGER. |
| COLUMN_VALUE_LONG procedure | Retrieves the value of column of type CLOB(32767). |
| COLUMN_VALUE_NUMBER procedure | Retrieves the value of column of type DECFLOAT. |
| COLUMN_VALUE_RAW procedure | Retrieves the value of column of type BLOB(32767). |
| COLUMN_VALUE_TIMESTAMP procedure | Retrieves the value of column of type TIMESTAMP |
| COLUMN_VALUE_VARCHAR procedure | Retrieves the value of column of type VARCHAR. |
| DEFINE_COLUMN_BLOB procedure | Defines the data type of the column to be BLOB. |
| DEFINE_COLUMN_CHAR procedure | Defines the data type of the column to be CHAR. |
| DEFINE_COLUMN_CLOB procedure | Defines the data type of the column to be CLOB. |
| DEFINE_COLUMN_DATE procedure | Defines the data type of the column to be DATE. |
| DEFINE_COLUMN_DOUBLE procedure | Defines the data type of the column to be DOUBLE. |
| DEFINE_COLUMN_INT procedure | Defines the data type of the column to be INTEGER. |
| DEFINE_COLUMN_LONG procedure | Defines the data type of the column to be CLOB(32767). |
| DEFINE_COLUMN_NUMBER procedure | Defines the data type of the column to be DECFLOAT. |
| DEFINE_COLUMN_RAW procedure | Defines the data type of the column to be BLOB(32767). |
| DEFINE_COLUMN_TIMESTAMP procedure | Defines the data type of the column to be TIMESTAMP. |
| DEFINE_COLUMN_VARCHAR procedure | Defines the data type of the column to be VARCHAR. |

Table 23. System-defined routines available in the DBMS_SQL module (continued)

| Procedure name | Description |
|---|---|
| DESCRIBE_COLUMNS procedure | Return a description of the columns retrieved by a cursor. |
| DESCRIBE_COLUMNS2 procedure | Identical to DESCRIBE_COLUMNS, but allows for column names greater than 32 characters. |
| EXECUTE procedure | Executes a cursor. |
| EXECUTE_AND_FETCH procedure | Executes a cursor and fetch one row. |
| FETCH_ROWS procedure | Fetches rows from a cursor. |
| IS_OPEN procedure | Checks if a cursor is open. |
| LAST_ROW_COUNT procedure | Returns the total number of rows fetched. |
| OPEN_CURSOR procedure | Opens a cursor. |
| PARSE procedure | Parses a DDL statement. |
| VARIABLE_VALUE_BLOB procedure | Retrieves the value of INOUT or OUT parameters as BLOB. |
| VARIABLE_VALUE_CHAR procedure | Retrieves the value of INOUT or OUT parameters as CHAR. |
| VARIABLE_VALUE_CLOB procedure | Retrieves the value of INOUT or OUT parameters as CLOB. |
| VARIABLE_VALUE_DATE procedure | Retrieves the value of INOUT or OUT parameters as DATE. |
| VARIABLE_VALUE_DOUBLE procedure | Retrieves the value of INOUT or OUT parameters as DOUBLE. |
| VARIABLE_VALUE_INT procedure | Retrieves the value of INOUT or OUT parameters as INTEGER. |
| VARIABLE_VALUE_NUMBER procedure | Retrieves the value of INOUT or OUT parameters as DECFLOAT. |
| VARIABLE_VALUE_RAW procedure | Retrieves the value of INOUT or OUT parameters as BLOB(32767). |
| VARIABLE_VALUE_TIMESTAMP procedure | Retrieves the value of INOUT or OUT parameters as TIMESTAMP. |
| VARIABLE_VALUE_VARCHAR procedure | Retrieves the value of INOUT or OUT parameters as VARCHAR. |

The following table lists the system-defined types and constants available in the DBMS_SQL module.

Table 24. DBMS_SQL system-defined types and constants

| Name | Type or constant | Description |
|---|---|---|
| DESC_REC | Type | A record of column information. |
| DESC_REC2 | Type | A record of column information. |
| DESC_TAB | Type | An array of records of type DESC_REC. |
| DESC_TAB2 | Type | An array of records of type DESC_REC2. |

Table 24. DBMS_SQL system-defined types and constants (continued)

| Name | Type or constant | Description |
|---|---|---|
| NATIVE | Constant | The only value supported for language_flag parameter of the PARSE procedure. |

### Usage notes

The routines in the DBMS_SQL module are useful when you want to construct and run dynamic SQL statements. For example, you might want execute DDL or DML statements such as "ALTER TABLE" or "DROP TABLE", construct and execute SQL statements on the fly, or call a function which uses dynamic SQL from within a SQL statement.

## BIND_VARIABLE_BLOB procedure - Bind a BLOB value to a variable

The BIND_VARIABLE_BLOB procedure provides the capability to associate a BLOB value with an IN, INOUT, or OUT argument in an SQL command.

### Syntax

```
►►—BIND_VARIABLE_BLOB—(—c—,—name—,—value—)——————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
An input argument of type BLOB(2G) that specifies the value to be assigned.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## BIND_VARIABLE_CHAR procedure - Bind a CHAR value to a variable

The BIND_VARIABLE_CHAR procedure provides the capability to associate a CHAR value with an IN, INOUT, or OUT argument in an SQL command.

### Syntax

```
►►—BIND_VARIABLE_CHAR—(—c—,—name—,—value—┬———————————————————┬—)————————►◄
                                         └—,—out_value_size—┘
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
An input argument of type CHAR(254) that specifies the value to be assigned.

*out_value_size*
An optional input argument of type INTEGER that specifies the length limit for the IN or INOUT argument, and the maximum length of the output value for the INOUT or OUT argument. If it is not specified, the length of *value* is assumed.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## BIND_VARIABLE_CLOB procedure - Bind a CLOB value to a variable

The BIND_VARIABLE_CLOB procedure provides the capability to associate a CLOB value with an IN, INOUT, or OUT argument in an SQL command.

### Syntax

```
►►──BIND_VARIABLE_CLOB──(──c──,──name──,──value──)─────────────────────────►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
An input argument of type CLOB(2G) that specifies the value to be assigned.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## BIND_VARIABLE_DATE procedure - Bind a DATE value to a variable

The BIND_VARIABLE_DATE procedure provides the capability to associate a DATE value with an IN, INOUT, or OUT argument in an SQL command.

### Syntax

```
►►──BIND_VARIABLE_DATE──(──c──,──name──,──value──)─────────────────────────►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
An input argument of type DATE that specifies the value to be assigned.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## BIND_VARIABLE_DOUBLE procedure - Bind a DOUBLE value to a variable

The BIND_VARIABLE_DOUBLE procedure provides the capability to associate a DOUBLE value with an IN, INOUT, or OUT argument in an SQL command.

### Syntax

```
►►──BIND_VARIABLE_DOUBLE──(──c──,──name──,──value──)──────────────────────►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
An input argument of type DOUBLE that specifies the value to be assigned.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## BIND_VARIABLE_INT procedure - Bind an INTEGER value to a variable

The BIND_VARIABLE_INT procedure provides the capability to associate an INTEGER value with an IN or INOUT bind variable in an SQL command.

### Syntax

```
►►──BIND_VARIABLE_INT──(──c──,──name──,──value──)─────────────────────────►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
An input argument of type INTEGER that specifies the value to be assigned.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## BIND_VARIABLE_NUMBER procedure - Bind a NUMBER value to a variable

The BIND_VARIABLE_NUMBER procedure provides the capability to associate a NUMBER value with an IN, INOUT, or OUT argument in an SQL command.

### Syntax

```
►►—BIND_VARIABLE_NUMBER—(—c—,—name—,—value—)——————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
An input argument of type DECFLOAT that specifies the value to be assigned.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## BIND_VARIABLE_RAW procedure - Bind a RAW value to a variable

The BIND_VARIABLE_RAW procedure provides the capability to associate a RAW value with an IN, INOUT, or OUT argument in an SQL command.

### Syntax

```
►►—BIND_VARIABLE_RAW—(—c—,—name—,—value—┬———————————————————┬—)————————►◄
                                        └—,—out_value_size—┘
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
An input argument of type BLOB(32767) that specifies the value to be assigned.

*out_value_size*
An optional input argument of type INTEGER that specifies the length limit

for the IN or INOUT argument, and the maximum length of the output value for the INOUT or OUT argument. If it is not specified, the length of *value* is assumed.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## BIND_VARIABLE_TIMESTAMP procedure - Bind a TIMESTAMP value to a variable

The BIND_VARIABLE_TIMESTAMP procedure provides the capability to associate a TIMESTAMP value with an IN, INOUT, or OUT argument in an SQL command.

Syntax

```
►►─BIND_VARIABLE_TIMESTAMP─(─c─,─name─,─value─)──────────────────────►◄
```

### Parameters

*c*
: An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
: An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
: An input argument of type TIMESTAMP that specifies the value to be assigned.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## BIND_VARIABLE_VARCHAR procedure - Bind a VARCHAR value to a variable

The BIND_VARIABLE_VARCHAR procedure provides the capability to associate a VARCHAR value with an IN, INOUT, or OUT argument in an SQL command.

### Syntax

```
►►─BIND_VARIABLE_VARCHAR─(─c─,─name─,─value──┬───────────────────┬─)────►◄
                                             └─,─out_value_size─┘
```

### Parameters

*c*
: An input argument of type INTEGER that specifies the cursor ID for the SQL command with bind variables.

*name*
: An input argument of type VARCHAR(128) that specifies the name of the bind variable in the SQL command.

*value*
An input argument of type VARCHAR(32672) that specifies the value to be assigned.

*out_value_size*
An input argument of type INTEGER that specifies the length limit for the IN or INOUT argument, and the maximum length of the output value for the INOUT or OUT argument. If it is not specified, the length of *value* is assumed.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## CLOSE_CURSOR procedure - Close a cursor

The CLOSE_CURSOR procedure closes an open cursor. The resources allocated to the cursor are released and it cannot no longer be used.

### Syntax

```
►►──CLOSE_CURSOR──(──c──)──────────────────────────────────────────────────►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor to be closed.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

### Examples

*Example 1:* This example illustrates closing a previously opened cursor.

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
            .
            .
            .
    DBMS_SQL.CLOSE_CURSOR(curid);
END;
```

## COLUMN_VALUE_BLOB procedure - Return a BLOB column value into a variable

The COLUMN_VALUE_BLOB procedure defines a variable that will receive a BLOB value from a cursor.

### Syntax

```
►►──COLUMN_VALUE_BLOB──(──c──,──position──,──value─────────────────────────►◄
```

### Parameters

*c*
An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type BLOB(2G) that specifies the variable receiving the data returned by the cursor in a prior fetch call.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_CHAR procedure - Return a CHAR column value into a variable

The COLUMN_VALUE_CHAR procedure defines a variable to receive a CHAR value from a cursor.

### Syntax

```
►►──COLUMN_VALUE_CHAR──(──c──,──position──,──value──────────────────────────►

►──┬───────────────────────────────────────────┬──)───────────────────────────►◄
   └─,──column_error──┬─────────────────────┬──┘
                      └─,──actual_length────┘
```

### Parameters

*c*
An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type CHAR that specifies the variable receiving the data returned by the cursor in a prior fetch call.

*column_error*
An optional output argument of type INTEGER that returns the SQLCODE, if any, associated with the column.

*actual_length*
An optional output argument of type INTEGER that returns the actual length of the data, prior to any truncation.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_CLOB procedure - Return a CLOB column value into a variable

The COLUMN_VALUE_CLOB procedure defines a variable that will receive a CLOB value from a cursor.

### Syntax

```
►►—COLUMN_VALUE_CLOB—(—c—,—position—,—value——————————————————————►◄
```

### Parameters

*c*
An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type CLOB(2G) that specifies the variable receiving the data returned by the cursor in a prior fetch call.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_DATE procedure - Return a DATE column value into a variable

The COLUMN_VALUE_DATE procedure defines a variable that will receive a DATE value from a cursor.

### Syntax

```
►►—COLUMN_VALUE_DATE—(—c—,—position—,—value——————————————————————►

►—┬——————————————————————————————————————┬—)——————————————————————►◄
  └—,—column_error—┬————————————————————┬┘
                   └—,—actual_length—┘
```

### Parameters

*c*
An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type DATE that specifies the variable receiving the data returned by the cursor in a prior fetch call.

*column_error*
An output argument of type INTEGER that returns the SQLCODE, if any, associated with the column.

*actual_length*
An output argument of type INTEGER that returns the actual length of the data, prior to any truncation.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_DOUBLE procedure - Return a DOUBLE column value into a variable

The COLUMN_VALUE_DOUBLE procedure defines a variable that will receive a DOUBLE value from a cursor.

### Syntax

```
►►—COLUMN_VALUE_DOUBLE—(—c—,—position—,—value——————————————————————————►

►—┬——————————————————————————————————————┬—)————————————————————————————►◄
  └—,—column_error—┬——————————————————┬—┘
                   └—,—actual_length—┘
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type DOUBLE that specifies the variable receiving the data returned by the cursor in a prior fetch call.

*column_error*
An output argument of type INTEGER that returns the SQLCODE, if any, associated with the column.

*actual_length*
An output argument of type INTEGER that returns the actual length of the data, prior to any truncation.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_INT procedure - Return an INTEGER column value into a variable

The COLUMN_VALUE_INT procedure defines a variable that will receive a INTEGER value from a cursor.

### Syntax

```
►►—COLUMN_VALUE_INT—(—c—,—position—,—value——————————————————————————————►
►─┬──────────────────────────────────────┬─)──────────────────────────►◄
  └─,─column_error─┬──────────────────┬─┘
                   └─,─actual_length─┘
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type INTEGER that specifies the variable receiving the data returned by the cursor in a prior fetch call.

*column_error*
An output argument of type INTEGER that returns the SQLCODE, if any, associated with the column.

*actual_length*
An output argument of type INTEGER that returns the actual length of the data, prior to any truncation.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_LONG procedure - Return a LONG column value into a variable

The COLUMN_VALUE_LONG procedure defines a variable that will receive a portion of a LONG value from a cursor.

### Syntax

```
►►─COLUMN_VALUE_LONG─(─c─,─position─,─length─,──────────────────────────►

►─offset─,─value─,─value_length─)──────────────────────────────────────►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*length*
An input argument of type INTEGER that specifies the desired number of bytes of the LONG data to retrieve beginning at *offset*.

*offset*
An input argument of type INTEGER that specifies the position within the LONG value to start data retrieval.

*value*
An output argument of type CLOB(32760) that specifies the variable receiving the data returned by the cursor in a prior fetch call.

*value_length*
An output argument of type INTEGER that returns the actual length of the data returned.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_NUMBER procedure - Return a DECFLOAT column value into a variable

The COLUMN_VALUE_NUMBER procedure defines a variable that will receive a DECFLOAT value from a cursor.

### Syntax

```
►►──COLUMN_VALUE_NUMBER──(──c──,──position──,──value─────────────────────────►

►──┬─────────────────────────────────────┬──)──────────────────────────────►◄
   └─,──column_error──┬─────────────────┬┘
                      └─,──actual_length─┘
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type DECFLOAT that specifies the variable receiving the data returned by the cursor in a prior fetch call.

*column_error*
An optional output argument of type INTEGER that returns the SQLCODE, if any, associated with the column.

*actual_length*
An optional output argument of type INTEGER that returns the actual length of the data, prior to any truncation.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_RAW procedure - Return a RAW column value into a variable

The COLUMN_VALUE_RAW procedure defines a variable that will receive a RAW value from a cursor.

### Syntax

```
►►─COLUMN_VALUE_RAW──(─c──,─position──,─value──────────────────────►

►─┬──────────────────────────────────────┬─)──────────────────────►◄
  └─,─column_error─┬──────────────────┬──┘
                   └─,─actual_length──┘
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type BLOB(32767) that specifies the variable receiving the data returned by the cursor in a prior fetch call.

*column_error*
An optional output argument of type INTEGER that returns the SQLCODE, if any, associated with the column.

*actual_length*
An optional output argument of type INTEGER that returns the actual length of the data, prior to any truncation.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_TIMESTAMP procedure - Return a TIMESTAMP column value into a variable

The COLUMN_VALUE_TIMESTAMP procedure defines a variable that will receive a TIMESTAMP value from a cursor.

### Syntax

```
►►─COLUMN_VALUE_TIMESTAMP──(─c──,─position──,─value────────────────►

►─┬──────────────────────────────────────┬─)──────────────────────►◄
  └─,─column_error─┬──────────────────┬──┘
                   └─,─actual_length──┘
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type TIMESTAMP that specifies the variable receiving the data returned by the cursor in a prior fetch call.

*column_error*
An output argument of type INTEGER that returns the SQLCODE, if any, associated with the column.

*actual_length*
An output argument of type INTEGER that returns the actual length of the data, prior to any truncation.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## COLUMN_VALUE_VARCHAR procedure - Return a VARCHAR column value into a variable

The COLUMN_VALUE_VARCHAR procedure defines a variable that will receive a VARCHAR value from a cursor.

### Syntax

```
►►──COLUMN_VALUE_VARCHAR──(──c──,──position──,──value──────────────────────────►

►──┬───────────────────────────────────────────┬──)──────────────────────────►◄
   └─,──column_error──┬──────────────────────┬─┘
                      └─,──actual_length─────┘
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor that is returning data to the variable being defined.

*position*
An input argument of type INTEGER that specifies the position of the returned data within the cursor. The first value in the cursor is position 1.

*value*
An output argument of type VARCHAR(32672) that specifies the variable receiving the data returned by the cursor in a prior fetch call.

*column_error*
An output argument of type INTEGER that returns the SQLCODE, if any, associated with the column.

*actual_length*
An output argument of type INTEGER that returns the actual length of the data, prior to any truncation.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_BLOB- Define a BLOB column in the SELECT list

The DEFINE_COLUMN_BLOB procedure defines a BLOB column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►—DEFINE_COLUMN_BLOB—(—c—,—position—,—column—)——————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type BLOB(2G).

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_CHAR procedure - Define a CHAR column in the SELECT list

The DEFINE_COLUMN_CHAR procedure defines a CHAR column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►—DEFINE_COLUMN_CHAR—(—c—,—position—,—column—,—column_size—)————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type CHAR(254).

*column_size*
An input argument of type INTEGER that specifies the maximum length of the returned data. Returned data exceeding *column_size* is truncated to *column_size* characters.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_CLOB - Define a CLOB column in the SELECT list

The DEFINE_COLUMN_CLOB procedure defines a CLOB column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►—DEFINE_COLUMN_CLOB—(—c—,—position—,—column—)————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type CLOB(2G).

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_DATE - Define a DATE column in the SELECT list

The DEFINE_COLUMN_DATE procedure defines a DATE column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►—DEFINE_COLUMN_DATE—(—c—,—position—,—column—)————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type DATE.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_DOUBLE - Define a DOUBLE column in the SELECT list

The DEFINE_COLUMN_DOUBLE procedure defines a DOUBLE column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►—DEFINE_COLUMN_DOUBLE—(—c—,—position—,—column—)——————————————————————►◄
```

### Parameters

*c*
An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type DOUBLE.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_INT- Define an INTEGER column in the SELECT list

The DEFINE_COLUMN_INT procedure defines an INTEGER column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►──DEFINE_COLUMN_INT──(──c──,──position──,──column──)─────────────────────►◄
```

### Parameters

*c*
An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type INTEGER.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_LONG procedure - Define a LONG column in the SELECT list

The DEFINE_COLUMN_LONG procedure defines a LONG column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►──DEFINE_COLUMN_LONG──(──c──,──position──────────────────────────────────►◄
```

### Parameters

*c*
An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_NUMBER procedure - Define a DECFLOAT column in the SELECT list

The DEFINE_COLUMN_NUMBER procedure defines a DECFLOAT column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►──DEFINE_COLUMN_NUMBER──(──c──,──position──,──column──)──────────────────────►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type DECFLOAT.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_RAW procedure - Define a RAW column or expression in the SELECT list

The DEFINE_COLUMN_RAW procedure defines a RAW column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►──DEFINE_COLUMN_RAW──(──c──,──position──,──column──,──column_size──)──────────►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type BLOB(32767).

*column_size*
An input argument of type INTEGER that specifies the maximum length of the returned data. Returned data exceeding *column_size* is truncated to *column_size* characters.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_TIMESTAMP - Define a TIMESTAMP column in the SELECT list

The DEFINE_COLUMN_TIMESTAMP procedure defines a TIMESTAMP column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►—DEFINE_COLUMN_TIMESTAMP—(—c—,—position—,—column—)——————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type TIMESTAMP.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DEFINE_COLUMN_VARCHAR procedure - Define a VARCHAR column in the SELECT list

The DEFINE_COLUMN_VARCHAR procedure defines a VARCHAR column or expression in the SELECT list that is to be returned and retrieved in a cursor.

### Syntax

```
►►—DEFINE_COLUMN_VARCHAR—(—c—,—position—,—column—,—column_size—)———————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor handle associated with the SELECT command.

*position*
An input argument of type INTEGER that specifies the position of the column or expression in the SELECT list that is being defined.

*column*
An input argument of type VARCHAR(32672).

*column_size*
An input argument of type INTEGER that specifies the maximum length of the returned data. Returned data exceeding *column_size* is truncated to *column_size* characters.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DESCRIBE_COLUMNS procedure - Retrieve a description of the columns in a SELECT list

The DESCRIBE_COLUMNS procedure provides the capability to retrieve a description of the columns in a SELECT list from a cursor.

### Syntax

```
►►—DESCRIBE_COLUMNS—(—c—,—col_cnt—,—desc_tab—)——————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor whose columns are to be described.

*col_cnt*
An output argument of type INTEGER that returns the number of columns in the SELECT list of the cursor.

*desc_tab*
An output argument of type DESC_TAB that describes the column metadata. The DESC_TAB array provides information on each column in the specified cursor.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

### Usage notes

This procedure requires a user temporary table space with a page size of 4K; otherwise it returns an SQL0286N error. You can create the user temporary table space with this command:

```
CREATE USER TEMPORARY TABLESPACE DBMS_SQL_TEMP_TBS
```

DESC_TAB is an array of DESC_REC records of column information:

*Table 25. DESC_TAB definition through DESC_REC records*

| **Record name** | **Description** |
|---|---|
| col_type | SQL data type as defined in Supported SQL data types in C and C++ embedded SQL applications. |
| col_max_len | Maximum length of the column. |
| col_name | Column name. |
| col_name_len | Length of the column name. |
| col_schema | Always NULL. |

*Table 25. DESC_TAB definition through DESC_REC records (continued)*

| Record name | Description |
|---|---|
| col_schema_name_len | Always NULL. |
| col_precision | Precision of the column as defined in the database. If col_type denotes a graphic or DBCLOB SQL data type, then this variable indicates the maximum number of double-byte characters the column can hold. |
| col_scale | Scale of the column as defined in the database (only applies to DECIMAL, NUMERIC, TIMESTAMP). |
| col_charsetid | Always NULL. |
| col_charsetform | Always NULL. |
| col_null_ok | Nullable indicator. This has a value of 1 if the column is nullable, otherwise, 0. |

```
ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_REC AS ROW
(
  col_type INTEGER,
  col_max_len INTEGER,
  col_name VARCHAR(128),
  col_name_len INTEGER,
  col_schema_name VARCHAR(128),
  col_schema_name_len INTEGER,
  col_precision INTEGER,
  col_scale INTEGER,
  col_charsetid INTEGER,
  col_charsetform INTEGER,
  col_null_ok INTEGER
);

ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_TAB AS DESC_REC ARRAY[INTEGER];
```

## Examples

*Example 1:* The following example describes the `empno`, `ename`, `hiredate`, and `sal` columns from the "EMP" table.

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE handle INTEGER;
  DECLARE col_cnt INTEGER;
  DECLARE col DBMS_SQL.DESC_TAB;
  DECLARE i INTEGER DEFAULT 1;
  DECLARE CUR1 CURSOR FOR S1;

  CALL DBMS_SQL.OPEN_CURSOR( handle );
  CALL DBMS_SQL.PARSE( handle,
      'SELECT empno, firstnme, lastname, salary
        FROM employee', DBMS_SQL.NATIVE );
  CALL DBMS_SQL.DESCRIBE_COLUMNS( handle, col_cnt, col );

  IF col_cnt > 0 THEN
    CALL DBMS_OUTPUT.PUT_LINE( 'col_cnt = ' || col_cnt );
    CALL DBMS_OUTPUT.NEW_LINE();
    fetchLoop: LOOP
      IF i > col_cnt THEN
        LEAVE fetchLoop;
      END IF;
      CALL DBMS_OUTPUT.PUT_LINE( 'i = ' || i );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_name = ' || col[i].col_name );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_name_len = ' ||
          NVL(col[i].col_name_len, 'NULL') );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_schema_name = ' ||
          NVL( col[i].col_schema_name, 'NULL' ) );

      IF col[i].col_schema_name_len IS NULL THEN
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_schema_name_len = NULL' );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_schema_name_len = ' ||
            col[i].col_schema_name_len);
      END IF;

      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_type = ' || col[i].col_type );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_max_len = ' || col[i].col_max_len );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_precision = ' || col[i].col_precision );
      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_scale = ' || col[i].col_scale );

      IF col[i].col_charsetid IS NULL THEN
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetid = NULL' );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetid = ' || col[i].col_charsetid );
      END IF;

      IF col[i].col_charsetform IS NULL THEN
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetform = NULL' );
      ELSE
        CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_charsetform = ' || col[i].col_charsetform );
      END IF;

      CALL DBMS_OUTPUT.PUT_LINE( 'col[i].col_null_ok = ' || col[i].col_null_ok );
      CALL DBMS_OUTPUT.NEW_LINE();
      SET i = i + 1;
    END LOOP;
  END IF;
END@

Output:
col_cnt = 4

i = 1
col[i].col_name = EMPNO
col[i].col_name_len = 5
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 452
col[i].col_max_len = 6
col[i].col_precision = 6
col[i].col_scale = 0
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 0

i = 2
col[i].col_name = FIRSTNME
col[i].col_name_len = 8
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 448
col[i].col_max_len = 12
col[i].col_precision = 12
col[i].col_scale = 0
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 0
```

```
i = 3
col[i].col_name = LASTNAME
col[i].col_name_len = 8
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 448
col[i].col_max_len = 15
col[i].col_precision = 15
col[i].col_scale = 0
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 0

i = 4
col[i].col_name = SALARY
col[i].col_name_len = 6
col[i].col_schema_name = NULL
col[i].col_schema_name_len = NULL
col[i].col_type = 484
col[i].col_max_len = 5
col[i].col_precision = 9
col[i].col_scale = 2
col[i].col_charsetid = NULL
col[i].col_charsetform = NULL
col[i].col_null_ok = 1
```

## DESCRIBE_COLUMNS2 procedure - Retrieve a description of column names in a SELECT list

The DESCRIBE_COLUMNS2 procedure provides the capability to retrieve a description of the columns in a SELECT list from a cursor.

### Syntax

```
►►──DESCRIBE_COLUMNS──(──c──,──col_cnt──,──desc_tab2──)──────────────────────────►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor whose columns are to be described.

*col_cnt*
An output argument of type INTEGER that returns the number of columns in the SELECT list of the cursor.

*desc_tab*
An output argument of type DESC_TAB2 that describes the column metadata. The DESC_TAB2 array provides information on each column in the specified cursor

### Authorization

EXECUTE privilege on the DBMS_SQL module.

### Usage notes

This procedure requires a user temporary table space with a page size of 4K; otherwise it returns an SQL0286N error. You can create the user temporary table space with this command:

```
CREATE USER TEMPORARY TABLESPACE DBMS_SQL_TEMP_TBS
```

DESC_TAB2 is an array of DESC_REC2 records of column information:

*Table 26. DESC_TAB2 definition through DESC_REC2 records*

| Record name | Description |
|---|---|
| col_type | SQL data type as defined in Supported SQL data types in C and C++ embedded SQL applications. |
| col_max_len | Maximum length of the column. |
| col_name | Column name. |
| col_name_len | Length of the column name. |
| col_schema | Always NULL. |
| col_schema_name_len | Always NULL. |
| col_precision | Precision of the column as defined in the database. If col_type denotes a graphic or DBCLOB SQL data type, then this variable indicates the maximum number of double-byte characters the column can hold. |
| col_scale | Scale of the column as defined in the database (only applies to DECIMAL, NUMERIC, TIMESTAMP). |
| col_charsetid | Always NULL. |
| col_charsetform | Always NULL. |
| col_null_ok | Nullable indicator. This has a value of 1 if the column is nullable, otherwise, 0. |

```
ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_REC2 AS ROW
(
col_type INTEGER,
col_max_len INTEGER,
col_name VARCHAR(128),
col_name_len INTEGER,
col_schema_name VARCHAR(128),
col_schema_name_len INTEGER,
col_precision INTEGER,
col_scale INTEGER,
col_charsetid INTEGER,
col_charsetform INTEGER,
col_null_ok INTEGER
);

ALTER MODULE SYSIBMADM.DBMS_SQL PUBLISH TYPE DESC_TAB2 AS DESC_REC2 ARRAY[INTEGER];
```

# EXECUTE procedure - Run a parsed SQL statement

The EXECUTE function executes a parsed SQL statement.

## Syntax

```
►►—EXECUTE—(—c—,—ret—)———————————————————————————————————————►◄
```

## Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the parsed SQL statement to be executed.

*ret* An output argument of type INTEGER that returns the number of rows processed if the SQL command is DELETE, INSERT, or UPDATE; otherwise it returns 0.

## Authorization

EXECUTE privilege on the DBMS_SQL module.

## Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## Examples

*Example 1:* The following anonymous block inserts a row into the "DEPT" table.

```
SET SERVEROUTPUT ON@

CREATE TABLE dept (
  deptno DECIMAL(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname  VARCHAR(14) NOT NULL,
  loc    VARCHAR(13),
  CONSTRAINT dept_dname_uq UNIQUE( deptno, dname )
)@

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE dept
( deptno DECIMAL(2) NOT NULL CONSTRAINT dept_pk PRIMARY KEY,
  dname  VARCHAR(14) NOT NULL,
  loc    VARCHAR(13),
  CONSTRAINT dept_dname_uq UNIQUE( deptno, dname ) )
DB20000I  The SQL command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO dept VALUES (50, ''HR'', ''LOS ANGELES'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
```

```
END
DB20000I  The SQL command completed successfully.

Number of rows processed: 1
```

## EXECUTE_AND_FETCH procedure - Run a parsed SELECT command and fetch one row

The EXECUTE_AND_FETCH procedure executes a parsed SELECT command and fetches one row.

### Syntax

```
►►──EXECUTE_AND_FETCH──(──c──┬──────────┬──,──ret──)──────────────────────►◄
                             └─,──exact─┘
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor id of the cursor for the SELECT command to be executed.

*exact*
An optional argument of type INTEGER. If set to 1, an exception is thrown if the number of rows in the result set is not exactly equal to 1. If set to 0, no exception is thrown. The default is 0. A NO_DATA_FOUND (SQL0100W) exception is thrown if *exact* is set to 1 and there are no rows in the result set. A TOO_MANY_ROWS (SQL0811N) exception is thrown if *exact* is set to 1 and there is more than one row in the result set.

*ret* An output argument of type INTEGER that returns 1 if a row was fetched successfully, 0 if there are no rows to fetch.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

### Examples

*Example 1:* The following stored procedure uses the EXECUTE_AND_FETCH function to retrieve one employee using the employee's name. An exception will be thrown if the employee is not found, or there is more than one employee with the same name.

```
SET SERVEROUTPUT ON@

CREATE TABLE emp (
  empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename    VARCHAR(10),
  job      VARCHAR(9),
  mgr      DECIMAL(4),
  hiredate TIMESTAMP(0),
  sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm     DECIMAL(7,2) )@

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)@
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)@
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)@
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)@
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)@

CREATE OR REPLACE PROCEDURE select_by_name(
IN p_ename ANCHOR TO emp.ename)
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno ANCHOR TO emp.empno;
  DECLARE v_hiredate ANCHOR TO emp.hiredate;
  DECLARE v_sal ANCHOR TO emp.sal;
  DECLARE v_comm ANCHOR TO emp.comm;
  DECLARE v_disp_date VARCHAR(10);
  DECLARE v_sql VARCHAR(120);
  DECLARE v_status INTEGER;
  SET v_sql = 'SELECT empno, hiredate, sal, NVL(comm, 0)
      FROM emp e WHERE ename = :p_ename ';
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.BIND_VARIABLE_VARCHAR(curid, ':p_ename', UPPER(p_ename));
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_comm);
  CALL DBMS_SQL.EXECUTE_AND_FETCH(curid, 1 /*True*/, v_status);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_comm);
  SET v_disp_date = TO_CHAR(v_hiredate, 'MM/DD/YYYY');
  CALL DBMS_OUTPUT.PUT_LINE('Number    : ' || v_empno);
  CALL DBMS_OUTPUT.PUT_LINE('Name      : ' || UPPER(p_ename));
  CALL DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
  CALL DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
  CALL DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@

CALL select_by_name( 'MARTIN' )@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE emp
 ( empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
   ename    VARCHAR(10),
   job      VARCHAR(9),
   mgr      DECIMAL(4),
   hiredate TIMESTAMP(0),
   sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
   comm     DECIMAL(7,2) )
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)
DB20000I  The SQL command completed successfully.
```

```
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE PROCEDURE select_by_name(
IN p_ename ANCHOR TO emp.ename)
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno ANCHOR TO emp.empno;
  DECLARE v_hiredate ANCHOR TO emp.hiredate;
  DECLARE v_sal ANCHOR TO emp.sal;
  DECLARE v_comm ANCHOR TO emp.comm;
  DECLARE v_disp_date VARCHAR(10);
  DECLARE v_sql VARCHAR(120);
  DECLARE v_status INTEGER;
  SET v_sql = 'SELECT empno, hiredate, sal, NVL(comm, 0)
      FROM emp e WHERE ename = :p_ename ';
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.BIND_VARIABLE_VARCHAR(curid, ':p_ename', UPPER(p_ename));
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_comm);
  CALL DBMS_SQL.EXECUTE_AND_FETCH(curid, 1 /*True*/, v_status);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 2, v_hiredate);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 3, v_sal);
  CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_comm);
  SET v_disp_date = TO_CHAR(v_hiredate, 'MM/DD/YYYY');
  CALL DBMS_OUTPUT.PUT_LINE('Number    : ' || v_empno);
  CALL DBMS_OUTPUT.PUT_LINE('Name      : ' || UPPER(p_ename));
  CALL DBMS_OUTPUT.PUT_LINE('Hire Date : ' || v_disp_date);
  CALL DBMS_OUTPUT.PUT_LINE('Salary    : ' || v_sal);
  CALL DBMS_OUTPUT.PUT_LINE('Commission: ' || v_comm);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

CALL select_by_name( 'MARTIN' )

  Return Status = 0

Number    : 7654
Name      : MARTIN
Hire Date : 09/28/1981
Salary    : 1250.00
Commission: 1400.00
```

## FETCH_ROWS procedure - Retrieve a row from a cursor

The FETCH_ROWS function retrieves a row from a cursor

### Syntax

```
►►—FETCH_ROWS—(—c—,—ret—)———————————————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor from which to fetch a row.

*ret* An output argument of type INTEGER that returns 1 if a row was fetched successfully, 0 if there are no rows to fetch.

## Authorization

EXECUTE privilege on the DBMS_SQL module.

## Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## Examples

*Example 1:* The following examples fetches the rows from the "EMP" table and displays the results.

```
SET SERVEROUTPUT ON@

CREATE TABLE emp (
  empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename    VARCHAR(10),
  job      VARCHAR(9),
  mgr      DECIMAL(4),
  hiredate TIMESTAMP(0),
  sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm     DECIMAL(7,2) )@

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)@
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)@
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)@
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)@
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)@

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
        COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
       ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(v_empno || '   ' ||
            RPAD(v_ename, 10) || '  ' || TO_CHAR(v_hiredate,
            'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
            '9,999.99') || ' ' || TO_CHAR(NVL(v_comm, 0),
            '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE emp (empno  DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
    ename    VARCHAR(10), job      VARCHAR(9), mgr      DECIMAL(4),
    hiredate TIMESTAMP(0),
    sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
    comm     DECIMAL(7,2) )
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)
DB20000I  The SQL command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
      COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
      ' || '--------');
  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,
        10) || '  ' || TO_CHAR(v_hiredate,
        'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
        '9,999.99') || ' ' || TO_CHAR(NVL(v_comm,
         0), '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00      0.00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00      0.00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
```

## IS_OPEN procedure - Check if a cursor is open

The IS_OPEN function provides the capability to test if the given cursor is open.

### Syntax

```
►►—IS_OPEN—(—c—,—ret—)———————————————————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor to be tested.

*ret* An output argument of type BOOLEAN that indicates if the specified file is open (TRUE) or closed (FALSE).

### Authorization

EXECUTE privilege on the DBMS_SQL module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## LAST_ROW_COUNT procedure - return the cumulative number of rows fetched

The LAST_ROW_COUNT procedure returns the number of rows that have been fetched.

## Syntax

```
►►──LAST_ROW_COUNT──(──ret──)──────────────────────────────────────────────►◄
```

## Parameters

*ret* An output argument of type INTEGER that returns the number of rows that have been fetched so far in the current session. A call to DBMS_SQL.PARSE resets the counter.

## Authorization

EXECUTE privilege on the DBMS_SQL module.

## Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## Examples

*Example 1:* The following example uses the LAST_ROW_COUNT procedure to display the total number of rows fetched in the query.

```
SET SERVEROUTPUT ON@

CREATE TABLE emp (
  empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
  ename    VARCHAR(10),
  job      VARCHAR(9),
  mgr      DECIMAL(4),
  hiredate TIMESTAMP(0),
  sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
  comm     DECIMAL(7,2) )@

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)@
INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)@
INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)@
INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)@
INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)@

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;
  DECLARE v_rowcount INTEGER;

  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
     COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
     ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(v_empno || '   ' || RPAD(v_ename,
       10) || '  ' || TO_CHAR(v_hiredate,
       'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
       '9,999.99') || ' ' || TO_CHAR(NVL(v_comm,
       0), '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.LAST_ROW_COUNT( v_rowcount );
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows: ' || v_rowcount);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE emp ( empno    DECIMAL(4) NOT NULL CONSTRAINT emp_pk PRIMARY KEY,
   ename    VARCHAR(10), job      VARCHAR(9),
   mgr      DECIMAL(4),
   hiredate TIMESTAMP(0),
   sal      DECIMAL(7,2) CONSTRAINT emp_sal_ck CHECK (sal > 0),
   comm     DECIMAL(7,2) )
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL)
DB20000I  The SQL command completed successfully.

INSERT INTO emp VALUES (7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400)
DB20000I  The SQL command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  DECLARE v_empno DECIMAL(4);
  DECLARE v_ename VARCHAR(10);
  DECLARE v_hiredate DATE;
  DECLARE v_sal DECIMAL(7, 2);
  DECLARE v_comm DECIMAL(7, 2);
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;
  DECLARE v_rowcount INTEGER;
  SET v_sql = 'SELECT empno, ename, hiredate, sal, ' || 'comm FROM emp';

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 1, v_empno);
  CALL DBMS_SQL.DEFINE_COLUMN_VARCHAR(curid, 2, v_ename, 10);
  CALL DBMS_SQL.DEFINE_COLUMN_DATE(curid, 3, v_hiredate);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 4, v_sal);
  CALL DBMS_SQL.DEFINE_COLUMN_NUMBER(curid, 5, v_comm);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('EMPNO  ENAME       HIREDATE    SAL
     COMM');
  CALL DBMS_OUTPUT.PUT_LINE('-----  ----------  ----------  --------
     ' || '--------');

  FETCH_LOOP: LOOP
    CALL DBMS_SQL.FETCH_ROWS(curid, v_status);

    IF v_status = 0 THEN
      LEAVE FETCH_LOOP;
    END IF;

    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 1, v_empno);
    CALL DBMS_SQL.COLUMN_VALUE_VARCHAR(curid, 2, v_ename);
    CALL DBMS_SQL.COLUMN_VALUE_DATE(curid, 3, v_hiredate);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 4, v_sal);
    CALL DBMS_SQL.COLUMN_VALUE_NUMBER(curid, 5, v_comm);
    CALL DBMS_OUTPUT.PUT_LINE(
      v_empno || '   ' || RPAD(v_ename, 10) || '  ' || TO_CHAR(v_hiredate,
      'yyyy-mm-dd') || ' ' || TO_CHAR(v_sal,
      '9,999.99') || ' ' || TO_CHAR(NVL(v_comm,
      0), '9,999.99'));
  END LOOP FETCH_LOOP;

  CALL DBMS_SQL.LAST_ROW_COUNT( v_rowcount );
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows: ' || v_rowcount);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.

EMPNO  ENAME       HIREDATE    SAL       COMM
-----  ----------  ----------  --------  --------
7369   SMITH       1980-12-17    800.00      0.00
7499   ALLEN       1981-02-20  1,600.00    300.00
7521   WARD        1981-02-22  1,250.00    500.00
7566   JONES       1981-04-02  2,975.00      0.00
7654   MARTIN      1981-09-28  1,250.00  1,400.00
Number of rows: 5
```

## OPEN_CURSOR procedure - Open a cursor

The OPEN_CURSOR procedure creates a new cursor.

A cursor must be used to parse and execute any dynamic SQL statement. Once a cursor has been opened, it can be used again with the same or different SQL statements. The cursor does not have to be closed and reopened in order to be used again.

### Syntax

```
►►—OPEN_CURSOR—(—c—)————————————————————————————————————————————►◄
```

### Parameters

*c* An output argument of type INTEGER that specifies the cursor ID of the newly created cursor.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

### Examples

*Example 1:* The following example creates a new cursor:

```
DECLARE
    curid           INTEGER;
BEGIN
    curid := DBMS_SQL.OPEN_CURSOR;
            .
            .
            .
END;
```

## PARSE procedure - Parse an SQL statement

The PARSE procedure parses an SQL statement.

If the SQL command is a DDL command, it is immediately executed and does not require running the EXECUTE procedure.

### Syntax

```
►►—PARSE—(—c—,—statement—,—language_flag—)——————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of an open cursor.

*statement*
The SQL statement to be parsed.

*language_flag*
This argument is provided for Oracle syntax compatibility. Use a value of 1 or `DBMS_SQL.native`.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## Examples

*Example 1:* The following anonymous block creates a table named `job`. Note that DDL statements are executed immediately by the PARSE procedure and do not require a separate EXECUTE step.

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE curid INTEGER;
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno DECIMAL(3),
     ' || 'jname VARCHAR(9))', DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

BEGIN
  DECLARE curid INTEGER;
  CALL DBMS_SQL.OPEN_CURSOR(curid);
  CALL DBMS_SQL.PARSE(curid, 'CREATE TABLE job (jobno DECIMAL(3), ' ||
    'jname VARCHAR(9))', DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.
```

*Example 2:* The following inserts two rows into the `job` table.

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO job VALUES (100, ''ANALYST'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  SET v_sql = 'INSERT INTO job VALUES (200, ''CLERK'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

This example results in the following output:

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(50);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'INSERT INTO job VALUES (100, ''ANALYST'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  SET v_sql = 'INSERT INTO job VALUES (200, ''CLERK'')';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.EXECUTE(curid, v_status);
  CALL DBMS_OUTPUT.PUT_LINE('Number of rows processed: ' || v_status);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
```

```
DB20000I  The SQL command completed successfully.

Number of rows processed: 1
Number of rows processed: 1
```

*Example 3:* The following anonymous block uses the DBMS_SQL module to execute a block containing two INSERT statements. Note that the end of the block contains a terminating semicolon, whereas in the prior examples, the individual INSERT statements did not have a terminating semicolon.

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(100);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'BEGIN ' || 'INSERT INTO job VALUES (300, ''MANAGER''); '
       || 'INSERT INTO job VALUES (400, ''SALESMAN''); ' || 'END;';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END@
```

This example results in the following output:

```
BEGIN
  DECLARE curid INTEGER;
  DECLARE v_sql VARCHAR(100);
  DECLARE v_status INTEGER;

  CALL DBMS_SQL.OPEN_CURSOR(curid);
  SET v_sql = 'BEGIN ' || 'INSERT INTO job VALUES (300, ''MANAGER''); ' ||
    'INSERT INTO job VALUES (400, ''SALESMAN''); ' || 'END;';
  CALL DBMS_SQL.PARSE(curid, v_sql, DBMS_SQL.native);
  CALL DBMS_SQL.CLOSE_CURSOR(curid);
END
DB20000I  The SQL command completed successfully.
```

## VARIABLE_VALUE_BLOB procedure - Return the value of a BLOB INOUT or OUT parameter

The VARIABLE_VALUE_BLOB procedure provides the capability to return the value of a BLOB INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_BLOB—(—c—,—name—,—value—)——————————————————————►◄
```

### Parameters

*c*
An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type BLOB(2G) that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## VARIABLE_VALUE_CHAR procedure - Return the value of a CHAR INOUT or OUT parameter

The VARIABLE_VALUE_CHAR procedure provides the capability to return the value of a CHAR INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_CHAR—(—c—,—name—,—value—)——————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type CHAR(254) that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## VARIABLE_VALUE_CLOB procedure - Return the value of a CLOB INOUT or OUT parameter

The VARIABLE_VALUE_CLOB procedure provides the capability to return the value of a CLOB INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_CLOB—(—c—,—name—,—value—)——————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type CLOB(2G) that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## VARIABLE_VALUE_DATE procedure - Return the value of a DATE INOUT or OUT parameter

The VARIABLE_VALUE_DATE procedure provides the capability to return the value of a DATE INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_DATE—(—c—,—name—,—value—)——————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type DATE that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## VARIABLE_VALUE_DOUBLE procedure - Return the value of a DOUBLE INOUT or OUT parameter

The VARIABLE_VALUE_DOUBLE procedure provides the capability to return the value of a DOUBLE INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_DOUBLE—(—c—,—name—,—value—)————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type DOUBLE that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## VARIABLE_VALUE_INT procedure - Return the value of an INTEGER INOUT or OUT parameter

The VARIABLE_VALUE_INT procedure provides the capability to return the value of a INTEGER INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_INT—(—c—,—name—,—value—)———————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type INTEGER that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## VARIABLE_VALUE_NUMBER procedure - Return the value of a DECFLOAT INOUT or OUT parameter

The VARIABLE_VALUE_NUMBER procedure provides the capability to return the value of a DECFLOAT INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_NUMBER—(—c—,—name—,—value—)————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type DECFLOAT that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## VARIABLE_VALUE_RAW procedure - Return the value of a BLOB(32767) INOUT or OUT parameter

The VARIABLE_VALUE_RAW procedure provides the capability to return the value of a BLOB(32767) INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_RAW—(—c—,—name—,—value—)———————————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type BLOB(32767) that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## VARIABLE_VALUE_TIMESTAMP procedure - Return the value of a TIMESTAMP INOUT or OUT parameter

The VARIABLE_VALUE_TIMESTAMP procedure provides the capability to return the value of a TIMESTAMP INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_TIMESTAMP—(—c—,—name—,—value—)————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type TIMESTAMP that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## VARIABLE_VALUE_VARCHAR procedure - Return the value of a VARCHAR INOUT or OUT parameter

The VARIABLE_VALUE_VARCHAR procedure provides the capability to return the value of a VARCHAR INOUT or OUT parameter.

### Syntax

```
►►—VARIABLE_VALUE_VARCHAR—(—c—,—name—,—value—)——————————————————————►◄
```

### Parameters

*c* An input argument of type INTEGER that specifies the cursor ID of the cursor returning a bind variable.

*name*
An input argument that specifies the name of the bind variable.

*value*
An output argument of type VARCHAR(32672) that specifies the variable receiving the value.

### Authorization

EXECUTE privilege on the DBMS_SQL module.

## DBMS_UTILITY module

The DBMS_UTILITY module provides various utility programs.

The schema for this module is SYSIBMADM.

The DBMS_UTILITY module includes the following routines.

*Table 27. System-defined routines available in the DBMS_UTILITY module*

| Routine Name | Description |
|---|---|
| ANALYZE_DATABASE procedure | Analyze database tables, clusters, and indexes. |
| ANALYZE_PART_OBJECT procedure | Analyze a partitioned table or partitioned index. |
| ANALYZE_SCHEMA procedure | Analyze schema tables, clusters, and indexes. |
| CANONICALIZE procedure | Canonicalizes a string (for example, strips off white space). |
| COMMA_TO_TABLE procedure | Convert a comma-delimited list of names to a table of names. |
| COMPILE_SCHEMA procedure | Compile programs in a schema. |
| DB_VERSION procedure | Get the database version. |
| EXEC_DDL_STATEMENT procedure | Execute a DDL statement. |
| GET_CPU_TIME function | Get the current CPU time. |
| GET_DEPENDENCY procedure | Get objects that depend on the given object. |
| GET_HASH_VALUE function | Compute a hash value. |
| GET_TIME function | Get the current time. |
| NAME_RESOLVE procedure | Resolve the given name. |
| NAME_TOKENIZE procedure | Parse the given name into its component parts. |
| TABLE_TO_COMMA procedure | Convert a table of names to a comma-delimited list. |
| VALIDATE procedure | Make an invalid database object valid. |

The following table lists the system-defined variables and types available in the DBMS_UTILITY module.

*Table 28. DBMS_UTILITY public variables*

| Public variables | Data type | Description |
|---|---|---|
| lname_array | TABLE | For lists of long names. |
| uncl_array | TABLE | For lists of users and names. |

The LNAME_ARRAY is for storing lists of long names including fully-qualified names.

```
ALTER MODULE SYSIBMADM.DBMS_UTILITY PUBLISH TYPE LNAME_ARRAY AS VARCHAR(4000) ARRAY[];
```

The UNCL_ARRAY is for storing lists of users and names.

```
ALTER MODULE SYSIBMADM.DBMS_UTILITY PUBLISH TYPE UNCL_ARRAY  AS VARCHAR(227)  ARRAY[];
```

## ANALYZE_DATABASE procedure - Gather statistics on tables, clusters, and indexes

The ANALYZE_DATABASE procedure provides the capability to gather statistics on tables, clusters, and indexes in the database.

### Syntax

```
►►—ANALYZE_DATABASE—(—method——————————————————————————————————————►

►——————————————————————————————————————————————————————————)———————►◄
  └,—estimate_rows————————————————————————————————————————┘
                   └,—estimate_percent———————————————————┘
                                      └,—method_opt—┘
```

### Parameters

*method*

An input argument of type VARCHAR(128) that specifies the type of analyze functionality to perform. Valid values are:

- ESTIMATE - gather estimated statistics based upon on either a specified number of rows in *estimate_rows* or a percentage of rows in *estimate_percent*;
- COMPUTE - compute exact statistics; or
- DELETE – delete statistics from the data dictionary.

*estimate_rows*

An optional input argument of type INTEGER that specifies the number of rows on which to base estimated statistics. One of *estimate_rows* or *estimate_percent* must be specified if the *method* is ESTIMATE. The default value is NULL.

*estimate_percent*

An optional input argument of type INTEGER that specifies the percentage of rows upon which to base estimated statistics. One of *estimate_rows* or *estimate_percent* must be specified if the *method* is ESTIMATE. The default value is NULL.

*method_opt*

An optional input argument of type VARCHAR(1024) that specifies the object types to be analyzed. Any combination of the following keywords are valid:

- [FOR TABLE]
- [FOR ALL [ INDEXED ] COLUMNS ] [ SIZE n ]
- [FOR ALL INDEXES]

The default is NULL.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

## ANALYZE_PART_OBJECT procedure - Gather statistics on a partitioned table or partitioned index

The ANALYZE_PART_OBJECT procedure provides the capability to gather statistics on a partitioned table or index.

### Syntax

```
►►──ANALYZE_PART_OBJECT──(──schema──,──object_name─────────────────────────────────────────►

►──┬──────────────────────────────────────────────────────────────────────────────┬──)──────►◄
   └─,──object_type──┬──────────────────────────────────────────────────────────┬─┘
                     └─,──command_type──┬────────────────────────────────────┬──┘
                                        └─,──command_opt──┬─────────────────┬┘
                                                          └─,──sample_clause┘
```

### Parameters

*schema*
An input argument of type VARCHAR(128) that specifies the schema name of the schema whose objects are to be analyzed.

*object_name*
An input argument of type VARCHAR(128) that specifies the name of the partitioned object to be analyzed.

*object_type*
An optional input argument of type CHAR that specifies the type of object to be analyzed. Valid values are:

- T – table;
- I – index.

The default is T.

*command_type*
An optional input argument of type CHAR that specifies the type of analyze functionality to perform. Valid values are:

- E - gather estimated statistics based upon on a specified number of rows or a percentage of rows in the *sample_clause* clause;
- C - compute exact statistics; or
- V - validate the structure and integrity of the partitions.

The default value is E.

*command_opt*
An optional input argument of type VARCHAR(1024) that specifies the options for the statistics calculation. For *command_type* E or C, this argument can be any combination of:

- [ FOR TABLE ]
- [ FOR ALL COLUMNS ]
- [ FOR ALL LOCAL INDEXES ]

For *command_type* V, this argument can be CASCADE if *object_type* is T. The default value is NULL.

*sample_clause*
An optional input argument of type VARCHAR(128). If *command_type* is E, this argument contains the following clause to specify the number of rows or percentage of rows on which to base the estimate.

```
SAMPLE n { ROWS | PERCENT }
```

The default value is `SAMPLE 5 PERCENT`.

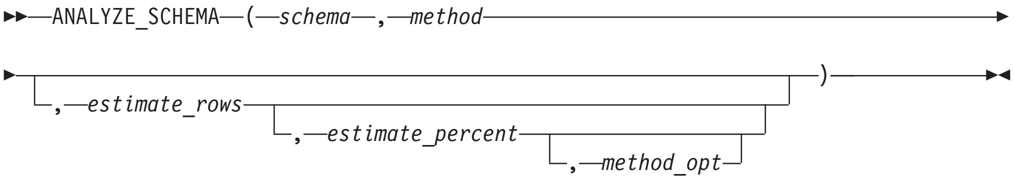
### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

## ANALYZE_SCHEMA procedure - Gather statistics on schema tables, clusters, and indexes

The ANALYZE_SCHEMA procedure provides the capability to gather statistics on tables, clusters, and indexes in the specified schema.

### Syntax



### Parameters

*schema*
An input argument of type VARCHAR(128) that specifies the schema name of the schema whose objects are to be analyzed.

*method*
An input argument of type VARCHAR(128) that specifies the type of analyze functionality to perform. Valid values are:

- ESTIMATE - gather estimated statistics based upon on either a specified number of rows in *estimate_rows* or a percentage of rows in *estimate_percent*;
- COMPUTE - compute exact statistics; or
- DELETE – delete statistics from the data dictionary.

*estimate_rows*
An optional input argument of type INTEGER that specifies the number of rows on which to base estimated statistics. One of *estimate_rows* or *estimate_percent* must be specified if the *method* is ESTIMATE. The default value is NULL.

*estimate_percent*
An optional input argument of type INTEGER that specifies the percentage of rows upon which to base estimated statistics. One of *estimate_rows* or *estimate_percent* must be specified if the *method* is ESTIMATE. The default value is NULL.

*method_opt*
An optional input argument of type VARCHAR(1024) that specifies the object types to be analyzed. Any combination of the following keywords are valid:

- [FOR TABLE]
- [FOR ALL [ INDEXED ] COLUMNS ] [ SIZE n ]
- [FOR ALL INDEXES]

The default is NULL.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

## CANONICALIZE procedure - Canonicalize a string

The CANONICALIZE procedure performs various operations on an input string.

The CANONICALIZE procedure performs the following operations on an input string:

- If the string is not double-quoted, verifies that it uses the characters of a legal identifier. If not, an exception is thrown. If the string is double-quoted, all characters are allowed.
- If the string is not double-quoted and does not contain periods, puts all alphabetic characters into uppercase and eliminates leading and trailing spaces.
- If the string is double-quoted and does not contain periods, strips off the double quotes.
- If the string contains periods and no portion of the string is double-quoted, puts each portion of the string into uppercase and encloses each portion in double quotes.
- If the string contains periods and portions of the string are double-quoted, returns the double-quoted portions unchanged, including the double quotes, and returns the non-double-quoted portions in uppercase and enclosed in double quotes.

### Syntax

```
►►—CANONICALIZE—(—name—,—canon_name—,—canon_len—)——————————————————————►◄
```

### Parameters

*name*
: An input argument of type VARCHAR(1024) that specifies the string to be canonicalized.

*canon_name*
: An output argument of type VARCHAR(1024) that returns the canonicalized string.

*canon_len*
: An input argument of type INTEGER that specifies the number of bytes in *name* to canonicalize starting from the first character.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

### Examples

*Example 1:* The following procedure applies the CANONICALIZE procedure on its input parameter and displays the results.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE canonicalize(
  IN p_name VARCHAR(4096),
  IN p_length INTEGER DEFAULT 30)
```

```
BEGIN
  DECLARE v_canon VARCHAR(100);

  CALL DBMS_UTILITY.CANONICALIZE(p_name, v_canon, p_length);
  CALL DBMS_OUTPUT.PUT_LINE('Canonicalized name ==>' || v_canon || '<==');
  CALL DBMS_OUTPUT.PUT_LINE('Length: ' || LENGTH(v_canon));
END@

CALL canonicalize('Identifier')@
CALL canonicalize('"Identifier"')@
CALL canonicalize('"_+142%"')@
CALL canonicalize('abc.def.ghi')@
CALL canonicalize('"abc.def.ghi"')@
CALL canonicalize('"abc".def."ghi"')@
CALL canonicalize('"abc.def".ghi')@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE canonicalize(
  IN p_name VARCHAR(4096),
  IN p_length INTEGER DEFAULT 30)
BEGIN
  DECLARE v_canon VARCHAR(100);

  CALL DBMS_UTILITY.CANONICALIZE(p_name, v_canon, p_length);
  CALL DBMS_OUTPUT.PUT_LINE('Canonicalized name ==>' || v_canon || '<==');
  CALL DBMS_OUTPUT.PUT_LINE('Length: ' || LENGTH(v_canon));
END
DB20000I  The SQL command completed successfully.

CALL canonicalize('Identifier')

  Return Status = 0

Canonicalized name ==>IDENTIFIER<==
Length: 10

CALL canonicalize('"Identifier"')

  Return Status = 0

Canonicalized name ==>Identifier<==
Length: 10

CALL canonicalize('"_+142%"')

  Return Status = 0

Canonicalized name ==>_+142%<==
Length: 6

CALL canonicalize('abc.def.ghi')

  Return Status = 0

Canonicalized name ==>"ABC"."DEF"."GHI"<==
Length: 17

CALL canonicalize('"abc.def.ghi"')

  Return Status = 0

Canonicalized name ==>abc.def.ghi<==
Length: 11
```

```
CALL canonicalize('"abc".def."ghi"')

  Return Status = 0

Canonicalized name ==>"abc"."DEF"."ghi"<==
Length: 17

CALL canonicalize('"abc.def".ghi')

  Return Status = 0

Canonicalized name ==>"abc.def"."GHI"<==
Length: 15
```

## COMMA_TO_TABLE procedures - Convert a comma-delimited list of names into a table of names

The COMMA_TO_TABLE procedure converts a comma-delimited list of names into an array of names. Each entry in the list becomes an element in the array.

**Note:** The names must be formatted as valid identifiers.

### Syntax

```
►►—COMMA_TO_TABLE_LNAME—(—list—,—tablen—,—tab—)————————————————————►◄


►►—COMMA_TO_TABLE_UNCL—(—list—,—tablen—,—tab—)—————————————————————►◄
```

### Parameters

*list*
An input argument of type VARCHAR(32672) that specifies a comma-delimited list of names.

*tablen*
An output argument of type INTEGER that specifies the number of entries in *tab*.

*tab* An output argument of type LNAME_ARRAY or UNCL_ARRAY that contains a table of the individual names in *list*. See LNAME_ARRAY or UNCL_ARRAY for a description of *tab*.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

### Examples

*Example 1:* The following procedure uses the COMMA_TO_TABLE_LNAME procedure to convert a list of names to a table. The table entries are then displayed.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE comma_to_table(
  IN p_list VARCHAR(4096))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE loop_limit INTEGER;

    SET loop_limit = v_length;
    WHILE i <= loop_limit DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname[i]);
      SET i = i + 1;
    END WHILE;
  END;
END@

CALL comma_to_table('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE comma_to_table(
  IN p_list VARCHAR(4096))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE loop_limit INTEGER;

    SET loop_limit = v_length;
    WHILE i <= loop_limit DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname[i]);
      SET i = i + 1;
    END WHILE;
  END;
END
DB20000I  The SQL command completed successfully.

CALL comma_to_table('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')

  Return Status = 0

sample_schema.dept
sample_schema.emp
sample_schema.jobhist
```

## COMPILE_SCHEMA procedure - Compile all functions, procedures, triggers, and packages in a schema

The COMPILE_SCHEMA procedure provides the capability to recompile all functions, procedures, triggers, and packages in a schema.

### Syntax

```
►►──COMPILE_SCHEMA──(──schema──┬──────────────────────────────────────┬──)──────►◄
                               └─,──compile_all──┬──────────────────┬─┘
                                                 └─,──reuse_settings─┘
```

### Parameters

*schema*
An input argument of type VARCHAR(128) that specifies the schema in which the programs are to be recompiled.

*compile_all*
An optional input argument of type BOOLEAN that must be set to `false`, meaning that the procedure only recompiles programs currently in invalid state.

*reuse_settings*
An optional input argument of type BOOLEAN that must be set to `false`, meaning that the procedure uses the current session settings.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

## DB_VERSION procedure - Retrieve the database version

The DB_VERSION procedure returns the version number of the database.

### Syntax

```
►►──DB_VERSION──(──version──,──compatibility──)────────────────────────────────►◄
```

### Parameters

*version*
An output argument of type VARCHAR(1024) that returns the database version number.

*compatibility*
An output argument of type VARCHAR(1024) that returns the compatibility setting of the database.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

### Examples

*Example 1:* The following anonymous block displays the database version information.

```
SET SERVEROUTPUT ON@

BEGIN
  DECLARE v_version VARCHAR(80);
  DECLARE v_compat VARCHAR(80);

  CALL DBMS_UTILITY.DB_VERSION(v_version, v_compat);
  CALL DBMS_OUTPUT.PUT_LINE('Version: ' || v_version);
  CALL DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

BEGIN
  DECLARE v_version VARCHAR(80);
  DECLARE v_compat VARCHAR(80);

  CALL DBMS_UTILITY.DB_VERSION(v_version, v_compat);
  CALL DBMS_OUTPUT.PUT_LINE('Version: ' || v_version);
  CALL DBMS_OUTPUT.PUT_LINE('Compatibility: ' || v_compat);
END
DB20000I  The SQL command completed successfully.

Version: DB2 v9.7.0.0
Compatibility: DB2 v9.7.0.0
```

## EXEC_DDL_STATEMENT procedure - Run a DDL statement

The EXEC_DDL_STATEMENT procedure provides the capability to execute a DDL command.

### Syntax

```
►►──EXEC_DDL_STATEMENT──(──parse_string──)─────────────────────────────►◄
```

### Parameters

*parse_string*
An input argument of type VARCHAR(1024) that specifies the DDL command to execute.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

### Examples

*Example 1:* The following anonymous block creates the job table.

```
BEGIN
  CALL DBMS_UTILITY.EXEC_DDL_STATEMENT(
    'CREATE TABLE job (' ||
    'jobno DECIMAL(3),' ||
    'jname VARCHAR(9))' );
END@
```

## GET_CPU_TIME function - Retrieve the current CPU time

The GET_CPU_TIME function returns the CPU time in hundredths of a second from some arbitrary point in time.

### Syntax

```
►►──GET_CPU_TIME──(──)─────────────────────────────────────────────────►◄
```

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

### Examples

*Example 1:* The following SELECT command retrieves the current CPU time.

```
SELECT DBMS_UTILITY.GET_CPU_TIME FROM DUAL;

get_cpu_time
-------------
          603
```

## GET_DEPENDENCY procedure - List objects dependent on the given object

The GET_DEPENDENCY procedure provides the capability to list all objects that are dependent upon the given object.

### Syntax

```
►►──GET_DEPENDENCY──(──type──,──schema──,──name──)──────────────────────────────►◄
```

### Parameters

*type*
An input argument of type VARCHAR(128) that specifies the object type of *name*. Valid values are FUNCTION, INDEX, LOB, PACKAGE, PACKAGE BODY, PROCEDURE, SEQUENCE, TABLE, TRIGGER, and VIEW.

*schema*
An input argument of type VARCHAR(128) that specifies the name of the schema in which *name* exists.

*name*
An input argument of type VARCHAR(128) that specifies the name of the object for which dependencies are to be obtained.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

### Examples

*Example 1:* The following anonymous block finds dependencies on the table T1, and the function FUNC1.

```
SET SERVEROUTPUT ON@

CREATE TABLE SCHEMA1.T1 (C1 INTEGER)@

CREATE OR REPLACE FUNCTION SCHEMA2.FUNC1( parm1 INTEGER )
SPECIFIC FUNC1
RETURNS INTEGER
BEGIN
  RETURN parm1;
END@

CREATE OR REPLACE FUNCTION SCHEMA3.FUNC2()
SPECIFIC FUNC2
RETURNS INTEGER
BEGIN
  DECLARE retVal INTEGER;
  SELECT SCHEMA2.FUNC1(1) INTO retVal FROM SCHEMA1.T1;
END@
```

```
CALL DBMS_UTILITY.GET_DEPENDENCY('FUNCTION', 'SCHEMA2', 'FUNC1')@
CALL DBMS_UTILITY.GET_DEPENDENCY('TABLE', 'SCHEMA1', 'T1')@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE TABLE SCHEMA1.T1 (C1 INTEGER)
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE FUNCTION SCHEMA2.FUNC1( parm1 INTEGER )
SPECIFIC FUNC1
RETURNS INTEGER
BEGIN
  RETURN parm1;
END
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE FUNCTION SCHEMA3.FUNC2()
SPECIFIC FUNC2
RETURNS INTEGER
BEGIN
  DECLARE retVal INTEGER;
  SELECT SCHEMA2.FUNC1(1) INTO retVal FROM SCHEMA1.T1;
END
DB20000I  The SQL command completed successfully.

CALL DBMS_UTILITY.GET_DEPENDENCY('FUNCTION', 'SCHEMA2', 'FUNC1')

  Return Status = 0

DEPENDENCIES ON SCHEMA2.FUNC1
------------------------------------------------------------------
*FUNCTION SCHEMA2.FUNC1()
*   FUNCTION SCHEMA3 .FUNC2()

CALL DBMS_UTILITY.GET_DEPENDENCY('TABLE', 'SCHEMA1', 'T1')

  Return Status = 0

DEPENDENCIES ON SCHEMA1.T1
------------------------------------------------------------------
*TABLE SCHEMA1.T1()
*   FUNCTION SCHEMA3 .FUNC2()
```

## GET_HASH_VALUE function - Compute a hash value for a given string

The GET_HASH_VALUE function provides the capability to compute a hash value for a given string.

The function returns a generated hash value of type INTEGER, and the value is platform-dependent.

### Syntax

```
►►—GET_HASH_VALUE—(—name—,—base—,—hash_size—)——————————————————————►◄
```

### Parameters

*name*
An input argument of type VARCHAR(32672) that specifies the string for which a hash value is to be computed.

*base*
An input argument of type INTEGER that specifies the starting value at which hash values are to be generated.

*hash_size*
An input argument of type INTEGER that specifies the number of hash values for the desired hash table.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

### Examples

*Example 1:* The following example returns hash values for two strings. The starting value for the hash values is 100, with a maximum of 1024 distinct values.

```
SELECT DBMS_UTILITY.GET_HASH_VALUE('Peter',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1@
SELECT DBMS_UTILITY.GET_HASH_VALUE('Mary',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1@
```

This example results in the following output:

```
SELECT DBMS_UTILITY.GET_HASH_VALUE('Peter',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1

HASH_VALUE
--------------------
                 343

  1 record(s) selected.


SELECT DBMS_UTILITY.GET_HASH_VALUE('Mary',100,1024) AS HASH_VALUE FROM SYSIBM.SYSDUMMY1

HASH_VALUE
--------------------
                 760

  1 record(s) selected.
```

## GET_TIME function - Return the current time

The GET_TIME function provides the capability to return the current time in hundredths of a second.

### Syntax

```
►►—GET_TIME—(—)————————————————————————————————————————————————————►◄
```

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

### Examples

*Example 1:* The following example shows calls to the GET_TIME function.

```
SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
---------
  1555860

SELECT DBMS_UTILITY.GET_TIME FROM DUAL;

 get_time
---------
  1556037
```

## NAME_RESOLVE procedure - Obtain the schema and other membership information for a database object

The NAME_RESOLVE procedure provides the capability to obtain the schema and other membership information of a database object. Synonyms are resolved to their base objects.

### Syntax

```
►►—NAME_RESOLVE—(—name—,—context—,—schema—,—part1—,——————————————————►

►—part2—,—dblink—,—part1_type—,—object_number—)——————————————————————►◄
```

### Parameters

*name*
An input argument of type VARCHAR(1024) that specifies the name of the database object to resolve. Can be specified in the format:

`[[ a.]b.]c[@dblink ]`

*context*
An input argument of type INTEGER. Set to the following values:

- 1 - to resolve a function, procedure, or module name;
- 2 - to resolve a table, view, sequence, or synonym name; or
- 3 - to resolve a trigger name.

*schema*
An output argument of type VARCHAR(128) that specifies the name of the schema containing the object specified by *name*.

*part1*
An output argument of type VARCHAR(128) that specifies the name of the resolved table, view, sequence, trigger, or module.

*part2*
An output argument of type VARCHAR(128) that specifies the name of the resolved function or procedure (including functions and procedures within a module).

*dblink*
An output argument of type VARCHAR(128) that specifies name of the database link (if `@dblink` is specified in *name*).

*part1_type*
An output argument of type INTEGER. Returns the following values:

- 2 - resolved object is a table;
- 4 - resolved object is a view;

- 6 - resolved object is a sequence;
- 7 - resolved object is a stored procedure;
- 8 - resolved object is a stored function;
- 9 - resolved object is a module or a function or procedure within a module; or
- 12 - resolved object is a trigger.

*object_number*
: An output argument of type INTEGER that specifies the object identifier of the resolved database object.

## Authorization

EXECUTE privilege on the DBMS_UTILITY module.

## Examples

*Example 1:* The following stored procedure is used to display the returned values of the NAME_RESOLVE procedure for various database objects.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE name_resolve(
  IN p_name VARCHAR(4096),
  IN p_context DECFLOAT )
BEGIN
  DECLARE v_schema VARCHAR(30);
  DECLARE v_part1 VARCHAR(30);
  DECLARE v_part2 VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_part1_type DECFLOAT;
  DECLARE v_objectid DECFLOAT;

  CALL DBMS_UTILITY.NAME_RESOLVE(p_name, p_context, v_schema, v_part1, v_part2,
    v_dblink, v_part1_type, v_objectid);
  CALL DBMS_OUTPUT.PUT_LINE('name      : ' || p_name);
  CALL DBMS_OUTPUT.PUT_LINE('context   : ' || p_context);
  CALL DBMS_OUTPUT.PUT_LINE('schema    : ' || v_schema);
  IF v_part1 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part1     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part1     : ' || v_part1);
  END IF;
  IF v_part2 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part2     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part2     : ' || v_part2);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : ' || v_dblink);
  END IF;
  CALL DBMS_OUTPUT.PUT_LINE('part1 type: ' || v_part1_type);
  CALL DBMS_OUTPUT.PUT_LINE('object id : ' || v_objectid);
END@

DROP TABLE S1.T1@
CREATE TABLE S1.T1 (C1 INT)@

CREATE OR REPLACE PROCEDURE S2.PROC1
BEGIN
END@
```

```
CREATE OR REPLACE MODULE S3.M1@
ALTER MODULE S3.M1 PUBLISH FUNCTION F1() RETURNS BOOLEAN
BEGIN
  RETURN TRUE;
END@

CALL NAME_RESOLVE( 'S1.T1', 2 )@
CALL NAME_RESOLVE( 'S2.PROC1', 2 )@
CALL NAME_RESOLVE( 'S2.PROC1', 1 )@
CALL NAME_RESOLVE( 'PROC1', 1 )@
CALL NAME_RESOLVE( 'M1', 1 )@
CALL NAME_RESOLVE( 'S3.M1.F1', 1 )@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE name_resolve(
  IN p_name VARCHAR(4096),
  IN p_context DECFLOAT )
BEGIN
  DECLARE v_schema VARCHAR(30);
  DECLARE v_part1 VARCHAR(30);
  DECLARE v_part2 VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_part1_type DECFLOAT;
  DECLARE v_objectid DECFLOAT;

  CALL DBMS_UTILITY.NAME_RESOLVE(p_name, p_context, v_schema, v_part1, v_part2,
    v_dblink, v_part1_type, v_objectid);
  CALL DBMS_OUTPUT.PUT_LINE('name      : ' || p_name);
  CALL DBMS_OUTPUT.PUT_LINE('context   : ' || p_context);
  CALL DBMS_OUTPUT.PUT_LINE('schema    : ' || v_schema);
  IF v_part1 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part1     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part1     : ' || v_part1);
  END IF;
  IF v_part2 IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('part2     : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('part2     : ' || v_part2);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink    : ' || v_dblink);
  END IF;
  CALL DBMS_OUTPUT.PUT_LINE('part1 type: ' || v_part1_type);
  CALL DBMS_OUTPUT.PUT_LINE('object id : ' || v_objectid);
END
DB20000I  The SQL command completed successfully.

DROP TABLE S1.T1
DB20000I  The SQL command completed successfully.

CREATE TABLE S1.T1 (C1 INT)
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE PROCEDURE S2.PROC1
BEGIN
END
DB20000I  The SQL command completed successfully.

CREATE OR REPLACE MODULE S3.M1
```

```
DB20000I  The SQL command completed successfully.

ALTER MODULE S3.M1 PUBLISH FUNCTION F1() RETURNS BOOLEAN
BEGIN
  RETURN TRUE;
END
DB20000I  The SQL command completed successfully.

CALL NAME_RESOLVE( 'S1.T1', 2 )

  Return Status = 0

name      : S1.T1
context   : 2
schema    : S1
part1     : T1
part2     : NULL
dblink    : NULL
part1 type: 2
object id : 8

CALL NAME_RESOLVE( 'S2.PROC1', 2 )
SQL0204N  "S2.PROC1" is an undefined name.  SQLSTATE=42704

CALL NAME_RESOLVE( 'S2.PROC1', 1 )

  Return Status = 0

name      : S2.PROC1
context   : 1
schema    : S2
part1     : PROC1
part2     : NULL
dblink    : NULL
part1 type: 7
object id : 66611

CALL NAME_RESOLVE( 'PROC1', 1 )

  Return Status = 0

name      : PROC1
context   : 1
schema    : S2
part1     : NULL
part2     : PROC1
dblink    : NULL
part1 type: 7
object id : 66611

CALL NAME_RESOLVE( 'M1', 1 )

  Return Status = 0

name      : M1
context   : 1
schema    : S3
part1     : NULL
part2     : M1
dblink    : NULL
part1 type: 9
object id : 16

CALL NAME_RESOLVE( 'S3.M1.F1', 1 )

  Return Status = 0
```

```
name      : S3.M1.F1
context   : 1
schema    : S3
part1     : M1
part2     : F1
dblink    : NULL
part1 type: 9
object id : 16
```

*Example 2:* Resolve a table accessed by a database link. Note that NAME_RESOLVE does not check the validity of the database object on the remote database. It merely echoes back the components specified in the *name* argument.

```
BEGIN
    name_resolve('sample_schema.emp@sample_schema_link',2);
END;

name      : sample_schema.emp@sample_schema_link
context   : 2
schema    : SAMPLE_SCHEMA
part1     : EMP
part2     :
dblink    : SAMPLE_SCHEMA_LINK
part1 type: 0
object id : 0
```

## NAME_TOKENIZE procedure - Parse the given name into its component parts

The NAME_TOKENIZE procedure parses a name into its component parts. Names without double quotes are put into uppercase, and double quotes are stripped from names with double quotes.

### Syntax

```
►►—NAME_TOKENIZE—(—name—,—a—,—b—,—c—,—dblink—,—nextpos—)——————————►◄
```

### Parameters

*name*
: An input argument of type VARCHAR(1024) that specifies the string containing a name in the following format:

```
a[.b[.c]][@dblink ]
```

*a*
: An output argument of type VARCHAR(128) that returns the leftmost component.

*b*
: An output argument of type VARCHAR(128) that returns the second component, if any.

*c*
: An output argument of type VARCHAR(128) that returns the third component, if any.

*dblink*
: An output argument of type VARCHAR(32672) that returns the database link name.

*nextpos*
: An output argument of type INTEGER that specifies the position of the last character parsed in *name*.

## Authorization

EXECUTE privilege on the DBMS_UTILITY module.

## Examples

*Example 1:* The following stored procedure is used to display the returned values of the NAME_TOKENIZE procedure for various names.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE name_tokenize(
  IN p_name VARCHAR(100) )
BEGIN
  DECLARE v_a VARCHAR(30);
  DECLARE v_b VARCHAR(30);
  DECLARE v_c VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_nextpos INTEGER;

  CALL DBMS_UTILITY.NAME_TOKENIZE(p_name, v_a, v_b, v_c, v_dblink, v_nextpos);
  CALL DBMS_OUTPUT.PUT_LINE('name   : ' || p_name);
  IF v_a IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('a      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('a      : ' || v_a);
  END IF;
  IF v_b IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('b      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('b      : ' || v_b);
  END IF;
  IF v_c IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('c      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('c      : ' || v_c);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink : ' || v_dblink);
  END IF;
  IF v_nextpos IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
  END IF;
END@

CALL name_tokenize( 'b' )@
CALL name_tokenize( 'a.b' )@
CALL name_tokenize( '"a".b.c' )@
CALL name_tokenize( 'a.b.c@d' )@
CALL name_tokenize( 'a.b."c"@"d"' )@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE name_tokenize(
  IN p_name VARCHAR(100) )
BEGIN
  DECLARE v_a VARCHAR(30);
  DECLARE v_b VARCHAR(30);
  DECLARE v_c VARCHAR(30);
  DECLARE v_dblink VARCHAR(30);
  DECLARE v_nextpos INTEGER;

  CALL DBMS_UTILITY.NAME_TOKENIZE(p_name, v_a, v_b, v_c, v_dblink, v_nextpos);
  CALL DBMS_OUTPUT.PUT_LINE('name   : ' || p_name);
  IF v_a IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('a      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('a      : ' || v_a);
  END IF;
  IF v_b IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('b      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('b      : ' || v_b);
  END IF;
  IF v_c IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('c      : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('c      : ' || v_c);
  END IF;
  IF v_dblink IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('dblink : NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('dblink : ' || v_dblink);
  END IF;
  IF v_nextpos IS NULL THEN
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: NULL');
  ELSE
    CALL DBMS_OUTPUT.PUT_LINE('nextpos: ' || v_nextpos);
  END IF;
END
DB20000I  The SQL command completed successfully.

CALL name_tokenize( 'b' )

  Return Status = 0

name   : b
a      : B
b      : NULL
c      : NULL
dblink : NULL
nextpos: 1

CALL name_tokenize( 'a.b' )

  Return Status = 0

name   : a.b
a      : A
b      : B
c      : NULL
dblink : NULL
nextpos: 3

CALL name_tokenize( '"a".b.c' )

  Return Status = 0

name   : "a".b.c
a      : a
b      : B
c      : C
dblink : NULL
nextpos: 7

CALL name_tokenize( 'a.b.c@d' )
```

```
  Return Status = 0

name   : a.b.c@d
a      : A
b      : B
c      : C
dblink : D
nextpos: 7

CALL name_tokenize( 'a.b."c"@"d"' )

  Return Status = 0

name   : a.b."c"@"d"
a      : A
b      : B
c      : c
dblink : d
nextpos: 11
```

## TABLE_TO_COMMA procedures - Convert a table of names into a comma-delimited list of names

The TABLE_TO_COMMA procedures convert an array of names into a comma-delimited list of names. Each array element becomes a list entry.

**Note:** The names must be formatted as valid identifiers.

### Syntax

```
►►──TABLE_TO_COMMA_LNAME──(──tab──,──tablen──,──list──)──────────────────────►◄


►►──TABLE_TO_COMMA_UNCL──(──tab──,──tablen──,──list──)───────────────────────►◄
```

### Parameters

*tab* An input argument of type LNAME_ARRAY or UNCL_ARRAY that specifies the array containing names. See LNAME_ARRAY or UNCL_ARRAY for a description of *tab*.

*tablen*
An output argument of type INTEGER that returns the number of entries in *list*.

*list*
An output argument of type VARCHAR(32672) that returns the comma-delimited list of names from *tab*.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

### Examples

*Example 1:* The following example first uses the COMMA_TO_TABLE_LNAME procedure to convert a comma-delimited list to a table. The TABLE_TO_COMMA_LNAME procedure then converts the table back to a comma-delimited list which is displayed.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE table_to_comma(
  IN p_list VARCHAR(100))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  DECLARE v_listlen INTEGER;
  DECLARE v_list VARCHAR(80);

  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  CALL DBMS_OUTPUT.PUT_LINE('Table Entries');
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE LOOP_LIMIT INTEGER;
    SET LOOP_LIMIT = v_length;

    WHILE i <= LOOP_LIMIT DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname(i));
      SET i = i + 1;
    END WHILE;
  END;
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  CALL DBMS_UTILITY.TABLE_TO_COMMA_LNAME(r_lname, v_listlen, v_list);
  CALL DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END@

CALL table_to_comma('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')@
```

This example results in the following output:

```
SET SERVEROUTPUT ON
DB20000I  The SET SERVEROUTPUT command completed successfully.

CREATE OR REPLACE PROCEDURE table_to_comma(
  IN p_list VARCHAR(100))
BEGIN
  DECLARE r_lname DBMS_UTILITY.LNAME_ARRAY;
  DECLARE v_length INTEGER;
  DECLARE v_listlen INTEGER;
  DECLARE v_list VARCHAR(80);

  CALL DBMS_UTILITY.COMMA_TO_TABLE_LNAME(p_list, v_length, r_lname);
  CALL DBMS_OUTPUT.PUT_LINE('Table Entries');
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  BEGIN
    DECLARE i INTEGER DEFAULT 1;
    DECLARE LOOP_LIMIT INTEGER;
    SET LOOP_LIMIT = v_length;

    WHILE i <= LOOP_LIMIT DO
      CALL DBMS_OUTPUT.PUT_LINE(r_lname(i));
      SET i = i + 1;
    END WHILE;
  END;
  CALL DBMS_OUTPUT.PUT_LINE('-------------');
  CALL DBMS_UTILITY.TABLE_TO_COMMA_LNAME(r_lname, v_listlen, v_list);
  CALL DBMS_OUTPUT.PUT_LINE('Comma-Delimited List: ' || v_list);
END
DB20000I  The SQL command completed successfully.

CALL table_to_comma('sample_schema.dept,sample_schema.emp,sample_schema.jobhist')

  Return Status = 0

Table Entries
```

```
-------------
sample_schema.dept
sample_schema.emp
sample_schema.jobhist
-------------
Comma-Delimited List: sample_schema.dept,sample_schema.emp,sample_schema.jobhist
```

## VALIDATE procedure - Change an invalid routine into a valid routine

The VALIDATE procedure provides the capability to change the state of an invalid routine to valid.

### Syntax

```
►►──VALIDATE──(──object_id──)──────────────────────────────────────────────►◄
```

### Parameters

*object_id*
An input argument of type INTEGER that specifies the identifier of the routine to be changed to a valid state. The ROUTINEID column of the SYSCAT.ROUTINES view contains all the routine identifiers.

### Authorization

EXECUTE privilege on the DBMS_UTILITY module.

# MONREPORT module

The MONREPORT module provides a set of procedures for retrieving a variety of monitoring data and generating text reports.

The schema for this module is SYSIBMADM.

The MONREPORT module includes the following system-defined routines.

*Table 29. System-defined routines available in the MONREPORT module*

| Routine name | Description |
|---|---|
| "CONNECTION procedure - generate a report on connection metrics" on page 316 | The Connection report presents monitor data for each connection. |
| "CURRENTAPPS procedure - generate a report of point-in-time application processing metrics" on page 317 | The Current Applications report presents the current instantaneous state of processing of units of work, agents, and activities for each connection. The report starts with state information summed across connections, followed by a section for details for each connection. |
| "CURRENTSQL procedure - generate a report that summarizes activities" on page 317 | The Current SQL report lists the top activities currently running, as measured by various metrics. |
| "DBSUMMARY procedure - generate a summary report of system and application performance metrics" on page 318 | The Summary report contains in-depth monitor data for the entire database, as well as key performance indicators for each connection, workload, service class, and database member. |

*Table 29. System-defined routines available in the MONREPORT module (continued)*

| Routine name | Description |
|---|---|
| "LOCKWAIT procedure - generate a report of current lock waits" on page 319 | The Lock Waits report contains information about each lock wait currently in progress. Details include lock holder and requestor details, plus characteristics of the lock held and the lock requested. |
| "PKGCACHE procedure - generate a summary report of package cache metrics" on page 321 | The Package Cache report lists the top statements accumulated in the package cache as measured by various metrics. |

## Usage notes

Monitor element names are displayed in upper case (for example, TOTAL_CPU_TIME). To find out more information about a monitor element, search the *DB2 Information Center* for the monitor name.

For reports with a *monitoring_interval* input, negative values in a report are inaccurate. This may occur during a rollover of source data counters. To determine accurate values, re-run the report after the rollover is complete.

**Note:** The reports are implemented using SQL procedures within modules, and as such can be impacted by the package cache configuration. If you observe slow performance when running the reports, inspect your package cache configuration to ensure it is sufficient for your workload. For further information, see "pckcachesz - Package cache size configuration parameter".

The following examples demonstrate various ways to call the MONREPORT routines. The examples show the MONREPORT.CONNECTION(*monitoring_interval*, *application_handle*) procedure. You can handle optional parameters for which you do not want to enter a value in the following ways:

- You can always specify null or DEFAULT.
- For character inputs, you can specify an empty string (' ').
- If it is the last parameter, you can omit it.

To generate a report that includes a section for each connection, with the default monitoring interval of 10 seconds, make the following call to the MONREPORT.CONNECTION procedure:

```
call monreport.connection()
```

To generate a report that includes a section only for the connection with application handle 32, with the default monitoring interval of 10 seconds, you can make either of the following calls to the MONREPORT.CONNECTION procedure:

```
call monreport.connection(DEFAULT, 32)
```

```
call monreport.connection(10, 32)
```

To generate a report that includes a section for each connection, with a monitoring interval of 60 seconds, you can make either of the following calls to the MONREPORT.CONNECTION procedure:

```
call monreport.connection(60)
```

```
call monreport.connection(60, null)
```

By default, the reports in this module are generated in English. To change the language in which the reports are generated, change the CURRENT LOCALE LC_MESSAGES special register. For example, to generate the CONNECTION report in French, issue the following commands:

```
SET CURRENT LOCALE LC_MESSAGES = 'CLDR 1.5:fr_FR'
CALL MONREPORT.CONNECTION
```

## CONNECTION procedure - generate a report on connection metrics

The CONNECTION procedure gathers monitor data for each connection and produces a text-formatted report.

The CONNECTION procedure is available starting with DB2 Version 9.7 Fix Pack 1.

### Syntax

```
►►──CONNECTION──(──monitoring_interval──,──application_handle──)─────────────────►◄
```

### Parameters

*monitoring_interval*
An optional input argument of type INTEGER that specifies the duration in seconds that monitoring data is collected before it is reported. For example, if you specify a monitoring interval of 30, the routine calls table functions, waits 30 seconds and calls the table functions again. The routine then calculates the difference, which reflects changes during the interval. If the *monitoring_interval* argument is not specified (or if null is specified), the default value is `10`. The range of valid inputs are the integer values `0-3600` (that is, up to 1 hour).

*application_handle*
An optional input argument of type BIGINT that specifies an application handle that identifies a connection. If the *application_handle* argument is not specified (or if null is specified), the report includes a section for each connection. The default is null.

### Authorization

The following privilege is required:

- EXECUTE privilege on the MONREPORT module

The following examples demonstrate various ways to call the CONNECTION procedure. The first example produces a report for all connections with data displayed corresponding to an interval of 10 seconds:

```
call monreport.connection;
```

The next example produces a report for all connections with data displayed corresponding to interval of 30 seconds:

```
call monreport.connection(30);
```

The next example produces a report for a connection with an application handle of 34. Data is displayed based on absolute totals accumulated in the source table functions (rather than based on the current interval).

```
call monreport.connection(0, 34);
```

The next example produces a report for a connection with an application handle of 34. Data is displayed corresponding to an interval of 10 seconds.

```
call monreport.connection(DEFAULT, 34);
```

## CURRENTAPPS procedure - generate a report of point-in-time application processing metrics

The CURRENTAPPS procedure gathers information about the current instantaneous state of processing of units or work, agents, and activities for each connection.

The CURRENTAPPS procedure is available starting with DB2 Version 9.7 Fix Pack 1.

### Syntax

```
►►──CURRENTAPPS──(──)──────────────────────────────────────────────────►◄
```

### Authorization

The following privilege is required:

- EXECUTE privilege on the MONREPORT module

The following examples demonstrate ways to call the CURRENTAPPS procedure:

```
call monreport.currentapps;
call monreport.currentapps();
```

## CURRENTSQL procedure - generate a report that summarizes activities

The CURRENTSQL procedure generates a text-formatted report that summarizes currently running activities.

The CURRENTSQL procedure is available starting with DB2 Version 9.7 Fix Pack 1.

### Syntax

```
►►──CURRENTSQL──(──member──)───────────────────────────────────────────►◄
```

### Parameters

*member*
An input argument of type SMALLINT that determines whether to show data for a particular member or partition, or to show data summed across all members. If this argument is not specified (or if null is specified), the report shows values summed across all members. If a valid member number is specified, the report shows values for that member.

### Authorization

The following privilege is required:

- EXECUTE privilege on the MONREPORT module

The following examples demonstrate various ways to call the CURRENTSQL procedure. The first example produces a report that shows activity metrics aggregated across all members:

```
call monreport.currentsql;
```

The next example produces a report that shows activity metrics specific to the activity performance on member number 4.

```
call monreport.currentsql(4);
```

## DBSUMMARY procedure - generate a summary report of system and application performance metrics

The DBSUMMARY procedure generates a text-formatted monitoring report that summarizes system and application performance metrics.

The DBSUMMARY procedure is available starting with DB2 Version 9.7 Fix Pack 1.

The DB Summary report contains in-depth monitor data for the entire database as well as key performance indicators for each connection, workload, service class, and database member.

### Syntax

```
►►──DBSUMMARY──(──monitoring_interval──)────────────────────────────────►◄
```

### Parameters

*monitoring_interval*

An optional input argument of type INTEGER that specifies the duration in seconds that monitoring data is collected before it is reported. For example, if you specify a monitoring interval of 30, the routine calls the table functions, waits 30 seconds and then calls the table functions again. The DBSUMMARY procedure then calculates the difference, which reflects changes during the interval. If the *monitoring_interval* argument is not specified (or if null is specified), the default value is 10. The range of valid inputs are the integer values 0-3600 (that is, up to 1 hour).

### Authorization

The following privilege is required:

- EXECUTE privilege on the MONREPORT module

The following examples demonstrate various ways to call the DBSUMMARY procedure. The first example produces a report that displays data corresponding to an interval of 10 seconds:

```
call monreport.dbsummary;
```

The next example produces a report that displays data corresponding to an interval of 30 seconds.

```
call monreport.dbsummary(30);
```

## LOCKWAIT procedure - generate a report of current lock waits

The Lock Waits report contains information about each lock wait currently in progress. Details include information about the lock holder and requestor and characteristics of the lock held and the lock requested.

The LOCKWAIT procedure is available starting with DB2 Version 9.7 Fix Pack 1.

### Syntax

```
►►—LOCKWAIT—(—)——————————————————————————————————————————————————————►◄
```

### Authorization

The following privilege is required:

- EXECUTE privilege on the MONREPORT module

The following examples demonstrate various ways to call the LOCKWAIT procedure:

```
call monreport.lockwait;
call monreport.lockwait();
```

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
Monitoring report - current lock waits
--------------------------------------------------------------------------------
Database:               SAMPLE
Generated:              08/28/2009 07:16:26

================================================================================
Part 1 - Summary of current lock waits

--------------------------------------------------------------------------------

      REQ_APPLICATION  LOCK_MODE  HLD_APPLICATION  LOCK_  LOCK_OBJECT_TYPE
#     HANDLE           REQUESTED  _HANDLE          MODE
----  ---------------  ---------  ---------------  -----  ----------------------
1     26               U          21               U      ROW
2     25               U          21               U      ROW
3     24               U          21               U      ROW
4     23               U          21               U      ROW
5     22               U          21               U      ROW
6     27               U          21               U      ROW

================================================================================

...


390 record(s) selected.

Return Status = 0
```

*Figure 1. Sample MONREPORT.LOCKWAIT output - summary section*

```
================================================================================
Part 2: Details for each current lock wait

lock wait #:1
--------------------------------------------------------------------------------

-- Lock details --

LOCK_NAME            = 04000500040000000000000052
LOCK_WAIT_START_TIME = 2009-08-28-07.15.31.013802
LOCK_OBJECT_TYPE     = ROW
TABSCHEMA            = TRIPATHY
TABNAME              = INVENTORY
ROWID                = 4
LOCK_STATUS          = W
LOCK_ATTRIBUTES      = 0000000000000000
ESCALATION           = N

-- Requestor and holder application details --

Attributes           Requestor                      Holder
-------------------  -----------------------------  ----------------------------
APPLICATION_HANDLE   26                             21
APPLICATION_ID       *LOCAL.tripathy.090828111531   *LOCAL.tripathy.090828111435
APPLICATION_NAME     java                           java
SESSION_AUTHID       TRIPATHY                       TRIPATHY
MEMBER               0                              0
LOCK_MODE            -                              U
LOCK_MODE_REQUESTED  U                              -

-- Lock holder current agents --

AGENT_TID            = 41
REQUEST_TYPE         = FETCH
EVENT_STATE          = IDLE
EVENT_OBJECT         = REQUEST
EVENT_TYPE           = WAIT
ACTIVITY_ID          =
UOW_ID               =

-- Lock holder current activities --

ACTIVITY_ID          = 1
UOW_ID               = 1
LOCAL_START_TIME     = 2009-08-28-07.14.31.079757
ACTIVITY_TYPE        = READ_DML
ACTIVITY_STATE       = IDLE

STMT_TEXT            =
select * from inventory for update

-- Lock requestor waiting agent and activity --

AGENT_TID            = 39
REQUEST_TYPE         = FETCH
ACTIVITY_ID          = 1
UOW_ID               = 1
LOCAL_START_TIME     = 2009-08-28-07.15.31.012935
ACTIVITY_TYPE        = READ_DML
ACTIVITY_STATE       = EXECUTING

STMT_TEXT            =
select * from inventory for update
```

*Figure 2. Sample MONREPORT.LOCKWAIT output - details section*

# PKGCACHE procedure - generate a summary report of package cache metrics

The Package Cache Summary report lists the top statements accumulated in the package cache as measured by various metrics.

The PKGCACHE procedure is available starting with DB2 Version 9.7 Fix Pack 1.

## Syntax

```
►►──PKGCACHE──(──cache_interval──,──section_type──,──member──)──────────────────►◄
```

## Parameters

*cache_interval*
An optional input argument of type INTEGER that specifies the report should only include data for package cache entries that have been updated in the past number of minutes specified by the *cache_interval* value. For example a *cache_interval* value of 60 produces a report based on package cache entries that have been updated in the past 60 minutes. Valid values are integers between `0` and `10080`, which supports an interval of up to 7 days. If the argument is not specified (or if null is specified), the report includes data for package cache entries regardless of when they were added or updated.

*section_type*
An optional input argument of type CHAR(1) that specifies whether the report should include data for static SQL, dynamic SQL, or both. If the argument is not specified (or if null is specified), the report includes data for both types of SQL. Valid values are: `d` or `D` (for dynamic) and `s` or `S` (for static).

*member*
An optional input argument of type SMALLINT that determines whether to show data for a particular member or partition, or to show data summed across all members. If this argument is not specified (or if null is specified), the report shows values summed across all members. If a valid member number is specified, the report shows values for that member.

## Authorization

The following privilege is required:

- EXECUTE privilege on the MONREPORT module

The following examples demonstrate various ways to call the PKGCACHE procedure. The first example produces a report based on all statements in the package cache, with data aggregated across all members:

```
call monreport.pkgcache;
```

The next example produces a report based on both dynamic and static statements in the package cache for which metrics have been updated within the last 30 minutes, with data aggregated across all members:

```
call monreport.pkgcache(30);
```

The next example produces a report based on all dynamic statements in the package cache, with data aggregated across all members:

```
call monreport.pkgcache(DEFAULT, 'd');
```

The next example produces a report based on both dynamic and static statements in the package cache for which metrics have been updated within the last 30 minutes, with data specific to a member number 4:

```
call db2monreport.pkgcache(30, DEFAULT, 4);
```

# UTL_DIR module

The UTL_DIR module provides a set of routines for maintaining directory aliases that are used with the UTL_FILE module.

**Note:** The UTL_DIR module does not issue any direct operating system calls, for example, the mkdir or rmdir commands. Maintenance of the physical directories is outside the scope of this module.

The schema for this module is SYSIBMADM.

The UTL_DIR module includes the following system-defined routines.

*Table 30. System-defined routines available in the UTL_DIR module*

| **Routine name** | **Description** |
|---|---|
| CREATE_DIRECTORY procedure | Creates a directory alias for the specified path. |
| CREATE_OR_REPLACE_DIRECTORY procedure | Creates or replaces a directory alias for the specified path. |
| DROP_DIRECTORY procedure | Drops the specified directory alias. |
| GET_DIRECTORY_PATH procedure | Gets the corresponding path for the specified directory alias. |

## CREATE_DIRECTORY procedure - Create a directory alias

The CREATE_DIRECTORY procedure creates a directory alias for the specified path.

Directory information is stored in SYSTOOLS.DIRECTORIES, which is created in the SYSTOOLSPACE when you first reference this module for each database.

### Syntax

```
►►—UTL_DIR.CREATE_DIRECTORY—(—alias—,—path—)———————————————————————————►◄
```

### Procedure parameters

*alias*
An input argument of type VARCHAR(128) that specifies the directory alias.

*path*
An input argument of type VARCHAR(1024) that specifies the path.

### Authorization

EXECUTE privilege on the UTL_DIR module.

### Example

Create a directory alias, and use it in a call to the UTL_FILE.FOPEN function.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  CALL UTL_DIR.CREATE_DIRECTORY('mydir', '/home/user/temp/mydir');
  SET v_filehandle = UTL_FILE.FOPEN('mydir',v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL DBMS_OUTPUT.PUT_LINE('Opened file: ' || v_filename);
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

This example results in the following output:

```
Opened file: myfile.csv
```

## CREATE_OR_REPLACE_DIRECTORY procedure - Create or replace a directory alias

The CREATE_OR_REPLACE_DIRECTORY procedure creates or replaces a directory alias for the specified path.

Directory information is stored in SYSTOOLS.DIRECTORIES, which is created in the SYSTOOLSPACE when you first reference this module for each database.

### Syntax

```
►►──UTL_DIR.CREATE_OR_REPLACE_DIRECTORY──(──alias──,──path──)──────────────────►◄
```

### Procedure parameters

*alias*
  An input argument of type VARCHAR(128) that specifies the directory alias.

*path*
  An input argument of type VARCHAR(1024) that specifies the path.

### Authorization

EXECUTE privilege on the UTL_DIR module.

### Example

*Example 1:* Create a directory alias. Because the directory already exists, an error occurs.

```
CALL UTL_DIR.CREATE_DIRECTORY('mydir', 'home/user/temp/empdir')@
```

This example results in the following output:

```
SQL0438N  Application raised error or warning with diagnostic text: "directory
alias already defined".  SQLSTATE=23505
```

*Example 2:* Create or replace a directory alias.

```
CALL UTL_DIR.CREATE_OR_REPLACE_DIRECTORY('mydir', 'home/user/temp/empdir')@
```

This example results in the following output:

```
  Return Status = 0
```

## DROP_DIRECTORY procedure - Drop a directory alias

The DROP_DIRECTORY procedure drops the specified directory alias.

### Syntax

```
►►—UTL_DIR.DROP_DIRECTORY—(—alias—)——————————————————————————————————►◄
```

### Procedure parameters

*alias*
An input argument of type VARCHAR(128) that specifies the directory alias.

### Authorization

EXECUTE privilege on the UTL_DIR module.

### Example

Drop the specified directory alias.

```
CALL UTL_DIR.DROP_DIRECTORY('mydir')@
```

This example results in the following output:

```
  Return Status = 0
```

## GET_DIRECTORY_PATH procedure - Get the path for a directory alias

The GET_DIRECTORY_PATH procedure returns the corresponding path for a directory alias.

### Syntax

```
►►—UTL_DIR.GET_DIRECTORY_PATH—(—alias—,—path—)———————————————————————►◄
```

### Procedure parameters

*alias*
An input argument of type VARCHAR(128) that specifies the directory alias.

*path*
An output argument of type VARCHAR(1024) that specifies the path that is defined for a directory alias.

### Authorization

EXECUTE privilege on the UTL_DIR module.

## Example

Get the path that is defined for a directory alias.

```
CALL UTL_DIR.GET_DIRECTORY_PATH('mydir', ? )@
```

This example results in the following output:

```
Value of output parameters
--------------------------
Parameter Name  : PATH
Parameter Value : home/rhoda/temp/mydir

Return Status = 0
```

# UTL_FILE module

The UTL_FILE module provides a set of routines for reading from and writing to files on the database server's file system.

The schema for this module is SYSIBMADM.

The UTL_FILE module includes the following system-defined routines and types.

*Table 31. System-defined routines available in the UTL_FILE module*

| Routine name | Description |
|---|---|
| FCLOSE procedure | Closes a specified file. |
| FCLOSE_ALL procedure | Closes all open files. |
| FCOPY procedure | Copies text from one file to another. |
| FFLUSH procedure | Flushes unwritten data to a file |
| FOPEN function | Opens a file. |
| FREMOVE procedure | Removes a file. |
| FRENAME procedure | Renames a file. |
| GET_LINE procedure | Gets a line from a file. |
| IS_OPEN function | Determines whether a specified file is open. |
| NEW_LINE procedure | Writes an end-of-line character sequence to a file. |
| PUT procedure | Writes a string to a file. |
| PUT_LINE procedure | Writes a single line to a file. |
| PUTF procedure | Writes a formatted string to a file. |
| UTL_FILE.FILE_TYPE | Stores a file handle. |

The following is a list of named conditions (these are called "exceptions" by Oracle) that an application can receive.

*Table 32. Named conditions for an application*

| Condition Name | Description |
|---|---|
| access_denied | Access to the file is denied by the operating system. |
| charsetmismatch | A file was opened using FOPEN_NCHAR, but later I/O operations used non-CHAR functions such as PUTF or GET_LINE. |

*Table 32. Named conditions for an application (continued)*

| Condition Name | Description |
|---|---|
| delete_failed | Unable to delete file. |
| file_open | File is already open. |
| internal_error | Unhandled internal error in the UTL_FILE module. |
| invalid_filehandle | File handle does not exist. |
| invalid_filename | A file with the specified name does not exist in the path. |
| invalid_maxlinesize | The MAX_LINESIZE value for FOPEN is invalid. It must be between 1 and 32672. |
| invalid_mode | The open_mode argument in FOPEN is invalid. |
| invalid_offset | The ABSOLUTE_OFFSET argument for FSEEK is invalid. It must be greater than 0 and less than the total number of bytes in the file. |
| invalid_operation | File could not be opened or operated on as requested. |
| invalid_path | The specified path does not exist or is not visible to the database |
| read_error | Unable to read the file. |
| rename_failed | Unable to rename the file. |
| write_error | Unable to write to the file. |

### Usage notes

To reference directories on the file system, use a directory alias. You can create a directory alias by calling the UTL_DIR.CREATE_DIRECTORY or UTL_DIR.CREATE_OR_REPLACE_DIRECTORY procedures. For example, `CALL UTL_DIR.CREATE_DIRECTORY('mydir', 'home/user/temp/mydir')@`.

The UTL_FILE module executes file operations by using the DB2 instance ID. Therefore, if you are opening a file, verify that the DB2 instance ID has the appropriate operating system permissions.

The UTL_FILE module is supported only in a non-partitioned database environment.

## FCLOSE procedure - Close an open file

The FCLOSE procedure closes a specified file.

### Syntax

```
►►—UTL_FILE.FCLOSE—(—file—)——————————————————————————————————————►◄
```

### Procedure parameters

*file* An input or output argument of type UTL_FILE.FILE_TYPE that contains the file handle. When the file is closed, this value is set to 0.

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

Open a file, write some text to the file, and then close the file.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to the file.');
  CALL UTL_FILE.FCLOSE(v_filehandle);
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      CALL DBMS_OUTPUT.PUT_LINE('Closed file: ' || v_filename);
    END IF;
END@

CALL proc1@
```

This example results in the following output:

```
Closed file: myfile.csv
```

## FCLOSE_ALL procedure - Close all open files

The FCLOSE_ALL procedure closes all open files. The procedure runs successfully even if there are no open files to close.

### Syntax

```
►►──UTL_FILE.FCLOSE_ALL──────────────────────────────────────────────►◄
```

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

Open a couple of files, write some text to the files, and then close all open files.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  v_filehandle2   UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  DECLARE  v_filename2     VARCHAR(20) DEFAULT 'myfile2.csv';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to a file.');
  SET v_filehandle2 = UTL_FILE.FOPEN(v_dirAlias,v_filename2,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle2 );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle2,'Some text to write to another file.');
  CALL UTL_FILE.FCLOSE_ALL;
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      CALL DBMS_OUTPUT.PUT_LINE(v_filename || ' is now closed.');
    END IF;
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle2 );
    IF isOpen != TRUE THEN
      CALL DBMS_OUTPUT.PUT_LINE(v_filename2 || ' is now closed.');
    END IF;
END@

CALL proc1@
```

This example results in the following output:

```
myfile.csv is now closed.
myfile2.csv is now closed.
```

## FCOPY procedure - Copy text from one file to another

The FCOPY procedure copies text from one file to another.

### Syntax

```
►►──UTL_FILE.FCOPY──(──location──,──filename──,──dest_dir──,──dest_file──────────►

►──┬────────────────────────────────┬──)─────────────────────────────────────────►◄
   └─,──start_line──┬─────────────┬─┘
                    └─,──end_line─┘
```

### Procedure parameters

*location*
An input argument of type VARCHAR(128) that specifies the alias of the directory that contains the source file.

*filename*
An input argument of type VARCHAR(255) that specifies the name of the source file.

*dest_dir*
An input argument of type VARCHAR(128) that specifies the alias of the destination directory.

*dest_file*
An input argument of type VARCHAR(255) that specifies the name of the destination file.

*start_line*
An optional input argument of type INTEGER that specifies the line number of the first line of text to copy in the source file. The default is 1.

*end_line*
: An optional input argument of type INTEGER that specifies the line number of the last line of text to copy in the source file. If this argument is omitted or null, the procedure continues copying all text through the end of the file.

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

Make a copy of a file, empfile.csv, that contains a comma-delimited list of employees from the emp table.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile       UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias      VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file      VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file     VARCHAR(20) DEFAULT 'empcopy.csv';
  DECLARE    v_empline       VARCHAR(200);
  CALL UTL_FILE.FCOPY(v_dirAlias,v_src_file,v_dirAlias,v_dest_file);
END@

CALL proc1@
```

This example results in the following output:

```
  Return Status = 0
```

The file copy, `empcopy.csv`, contains the following data:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220
20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300
30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060
50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214
60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580
70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893
90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380
100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

## FFLUSH procedure - Flush unwritten data to a file

The FFLUSH procedure forces unwritten data in the write buffer to be written to a file.

### Syntax

```
►►—UTL_FILE.FFLUSH—(—file—)————————————————————————————————————————►◄
```

### Procedure parameters

*file* An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

Flush each line after calling the NEW_LINE procedure.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL UTL_FILE.PUT(v_empfile_tgt,v_empline);
    CALL UTL_FILE.NEW_LINE(v_empfile_tgt);
    CALL UTL_FILE.FFLUSH(v_empfile_tgt);
  END LOOP;
  CALL DBMS_OUTPUT.PUT_LINE('Updated file: ' || v_dest_file);
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

This example results in the following output:

```
Updated file: empfilenew.csv
```

The updated file, empfilenew.csv, contains the following data:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

## FOPEN function - Open a file

The FOPEN function opens a file for I/O.

### Syntax

```
►►──UTL_FILE.FOPEN──(──location──,──filename──,──open_mode──┬──────────────────┬──)──────────►◄
                                                            └─,──max_linesize──┘
```

## Return value

This function returns a value of type UTL_FILE.FILE_TYPE that indicates the file handle of the opened file.

## Function parameters

*location*
An input argument of type VARCHAR(128) that specifies the alias of the directory that contains the file.

*filename*
An input argument of type VARCHAR(255) that specifies the name of the file.

*open_mode*
An input argument of type VARCHAR(10) that specifies the mode in which the file is opened:

| | |
|---|---|
| *a* | append to file |
| *r* | read from file |
| *w* | write to file |

*max_linesize*
An optional input argument of type INTEGER that specifies the maximum size of a line in characters. The default value is 1024 bytes. In read mode, an exception is thrown if an attempt is made to read a line that exceeds *max_linesize*. In write and append modes, an exception is thrown if an attempt is made to write a line that exceeds *max_linesize*. End-of-line character(s) do not count towards the line size.

## Authorization

EXECUTE privilege on the UTL_FILE module.

## Example

Open a file, write some text to the file, and then close the file.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL DBMS_OUTPUT.PUT_LINE('Opened file: ' || v_filename);
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to the file.');
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

This example results in the following output.

```
Opened file: myfile.csv
```

## FREMOVE procedure - Remove a file

The FREMOVE procedure removes a specified file from the system. If the file does not exist, this procedure throws an exception.

### Syntax

```
►►—UTL_FILE.FREMOVE—(—location—,—filename—)——————————————————————————►◄
```

### Procedure parameters

*location*
An input argument of type VARCHAR(128) that specifies the alias of the directory that contains the file.

*filename*
An input argument of type VARCHAR(255) that specifies the name of the file.

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

Remove the file `myfile.csv` from the system.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  CALL UTL_FILE.FREMOVE(v_dirAlias,v_filename);
  CALL DBMS_OUTPUT.PUT_LINE('Removed file: ' || v_filename);
END@

CALL proc1@
```

This example results in the following output:

```
Removed file: myfile.csv
```

## FRENAME procedure - Rename a file

The FRENAME procedure renames a specified file. Renaming a file effectively moves a file from one location to another.

### Syntax

```
►►—UTL_FILE.FRENAME—(—location—,—filename—,—dest_dir—,—dest_file—┬────────────┬—)——————►◄
                                                                  └—,—replace—┘
```

### Procedure parameters

*location*
An input argument of type VARCHAR(128) that specifies the alias of the directory that contains the file that you want to rename.

*filename*
An input argument of type VARCHAR(255) that specifies the name of the file that you want to rename.

*dest_dir*
An input argument of type VARCHAR(128) that specifies the alias of the destination directory.

*dest_file*
An input argument of type VARCHAR(255) that specifies the new name of the file.

*replace*
An optional input argument of type INTEGER that specifies whether to replace the file *dest_file* in the directory *dest_dir* if the file already exists:

*1* Replaces existing file.

*0* Throws an exception if the file already exists. This is the default if no value is specified for *replace*.

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

Rename a file, empfile.csv, that contains a comma-delimited list of employees from the emp table.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_dirAlias      VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file      VARCHAR(20) DEFAULT 'oldemp.csv';
  DECLARE    v_dest_file     VARCHAR(20) DEFAULT 'newemp.csv';
  DECLARE    v_replace       INTEGER DEFAULT 1;
  CALL UTL_FILE.FRENAME(v_dirAlias,v_src_file,v_dirAlias,
        v_dest_file,v_replace);
  CALL DBMS_OUTPUT.PUT_LINE('The file ' || v_src_file ||
    ' has been renamed to ' || v_dest_file);
END@

CALL proc1@
```

This example results in the following output:

```
The file oldemp.csv has been renamed to newemp.csv
```

## GET_LINE procedure - Get a line from a file

The GET_LINE procedure gets a line of text from a specified file. The line of text does not include the end-of-line terminator. When there are no more lines to read, the procedure throws a NO_DATA_FOUND exception.

### Syntax

```
►►──UTL_FILE.GET_LINE──(──file──,──buffer──)──────────────────────────────────►◄
```

### Procedure parameters

*file* An input argument of type UTL_FILE.FILE_TYPE that contains the file handle of the opened file.

*buffer*
An output argument of type VARCHAR(32672) that contains a line of text from the file.

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

Read through and display the records in the file empfile.csv.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile       UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias      VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_filename      VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_empline       VARCHAR(200);
  DECLARE    v_count         INTEGER DEFAULT 0;
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE 'ORANF'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile = UTL_FILE.FOPEN(v_dirAlias,v_filename,'r');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile, v_empline);
    IF SQLSTATE1 = 'ORANF' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL DBMS_OUTPUT.PUT_LINE(v_empline);
    SET v_count = v_count + 1;
  END LOOP;
  CALL DBMS_OUTPUT.PUT_LINE('End of file ' || v_filename || ' - ' || v_count
        || ' records retrieved');
  CALL UTL_FILE.FCLOSE(v_empfile);
END@

CALL proc1@
```

This example results in the following output:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092

End of file empfile.csv - 8 records retrieved
```

## IS_OPEN function - Determine whether a specified file is open

The IS_OPEN function determines whether a specified file is open.

### Syntax

```
►►──UTL_FILE.IS_OPEN──(──file──)──────────────────────────────────────────►◄
```

### Return value

This function returns a value of type BOOLEAN that indicates if the specified file is open (TRUE) or closed (FALSE).

### Function parameters

*file* An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

The following example demonstrates that before writing text to a file, you can call the IS_OPEN function to check if the file is open.

```
SET SERVEROUTPUT ON@

CREATE OR REPLACE PROCEDURE proc1()
BEGIN
  DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE  isOpen          BOOLEAN;
  DECLARE  v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE  v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  SET isOpen = UTL_FILE.IS_OPEN( v_filehandle );
    IF isOpen != TRUE THEN
      RETURN -1;
    END IF;
  CALL UTL_FILE.PUT_LINE(v_filehandle,'Some text to write to the file.');
  CALL DBMS_OUTPUT.PUT_LINE('Updated file: ' || v_filename);
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

This example results in the following output.

```
Updated file: myfile.csv
```

## NEW_LINE procedure - Write an end-of-line character sequence to a file

The NEW_LINE procedure writes an end-of-line character sequence to a specified file.

### Syntax

```
►►──UTL_FILE.NEW_LINE──(──file──┬─────────────┬──)─────────────────────────►◄
                                └─,──lines──┘
```

### Procedure parameters

*file* An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

*lines*
An optional input argument of type INTEGER that specifies the number of end-of-line character sequences to write to the file. The default is 1.

## Authorization

EXECUTE privilege on the UTL_FILE module.

## Example

Write a file that contains a triple-spaced list of employee records.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL UTL_FILE.PUT(v_empfile_tgt,v_empline);
    CALL UTL_FILE.NEW_LINE(v_empfile_tgt, 2);
  END LOOP;

  CALL DBMS_OUTPUT.PUT_LINE('Wrote to file: ' || v_dest_file);
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

This example results in the following output:

```
Wrote to file: empfilenew.csv
```

The file that is updated, `empfilenew.csv`, contains the following data:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220


20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300


30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060


50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214


60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580
```

```
70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

## PUT procedure - Write a string to a file

The PUT procedure writes a string to a specified file. No end-of-line character sequence is written at the end of the string.

### Syntax

```
►►──UTL_FILE.PUT──(──file──,──buffer──)──────────────────────────────────────►◄
```

### Procedure parameters

*file* An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

*buffer*
An input argument of type VARCHAR(32672) that specifies the text to write to the file.

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

Use the PUT procedure to add a string to a file and then use the NEW_LINE procedure to add an end-of-line character sequence.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    CALL UTL_FILE.PUT(v_empfile_tgt,v_empline);
    CALL UTL_FILE.NEW_LINE(v_empfile_tgt, 2);
  END LOOP;

  CALL DBMS_OUTPUT.PUT_LINE('Wrote to file: ' || v_dest_file);
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

This example results in the following output:

```
Wrote to file: empfilenew.csv
```

The updated file, empfilenew.csv, contains the following data:

```
10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220


20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300


30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060


50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214


60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580


70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893


90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380


100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

### Usage notes

After using the PUT procedure to add a string to a file, use the NEW_LINE procedure to add an end-of-line character sequence to the file.

## PUT_LINE procedure - Write a line of text to a file

The PUT_LINE procedure writes a line of text, including an end-of-line character sequence, to a specified file.

### Syntax

```
►►──UTL_FILE.PUT_LINE──(──file──,──buffer──)──────────────────────────────►◄
```

### Procedure parameters

*file* An input argument of type UTL_FILE.FILE_TYPE that contains the file handle of file to which the line is to be written.

*buffer*
An input argument of type VARCHAR(32672) that specifies the text to write to the file.

### Authorization

EXECUTE privilege on the UTL_FILE module.

### Example

Use the PUT_LINE procedure to write lines of text to a file.

```
CALL proc1@
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE    v_empfile_src    UTL_FILE.FILE_TYPE;
  DECLARE    v_empfile_tgt    UTL_FILE.FILE_TYPE;
  DECLARE    v_dirAlias       VARCHAR(50) DEFAULT 'empdir';
  DECLARE    v_src_file       VARCHAR(20) DEFAULT 'empfile.csv';
  DECLARE    v_dest_file      VARCHAR(20) DEFAULT 'empfilenew2.csv';
  DECLARE    v_empline        VARCHAR(200);
  DECLARE    v_count          INTEGER DEFAULT 0;
  DECLARE SQLCODE INTEGER DEFAULT 0;
  DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
  DECLARE SQLSTATE1 CHAR(5) DEFAULT '00000';
  DECLARE CONTINUE HANDLER FOR SQLSTATE '02000'SET SQLSTATE1 = SQLSTATE;

  SET v_empfile_src = UTL_FILE.FOPEN(v_dirAlias,v_src_file,'r');
  SET v_empfile_tgt = UTL_FILE.FOPEN(v_dirAlias,v_dest_file,'w');

  loop1: LOOP
    CALL UTL_FILE.GET_LINE(v_empfile_src,v_empline);
    IF SQLSTATE1 = '02000' THEN -- NO DATA FOUND
      LEAVE loop1;
    END IF;
    SET v_count = v_count + 1;
    CALL UTL_FILE.PUT(v_empfile_tgt,'Record ' || v_count || ': ');
    CALL UTL_FILE.PUT_LINE(v_empfile_tgt,v_empline);
  END LOOP;
  CALL DBMS_OUTPUT.PUT_LINE('End of file ' || v_src_file || ' - ' || v_count
    || ' records retrieved');
  CALL UTL_FILE.FCLOSE_ALL;
END@

CALL proc1@
```

This example results in the following output:

```
End of file empfile.csv - 8 records retrieved
```

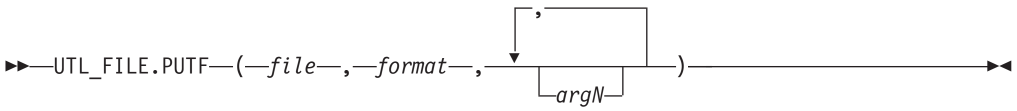
The file that is updated, `empfilenew2.csv`, contains the following data:

```
Record 1: 10,CHRISTINE,I,HAAS,A00,3978,1/1/1965,PRES,18,F,8/24/1933,52750,1000,4220

Record 2: 20,MICHAEL,L,THOMPSON,B01,3476,10/10/1973,MANAGER,18,M,2/2/1948,41250,800,3300

Record 3: 30,SALLY,A,KWAN,C01,4738,4/5/1975,MANAGER,20,F,5/11/1941,38250,800,3060

Record 4: 50,JOHN,B,GEYER,E01,6789,8/17/1949,MANAGER,16,M,9/15/1925,40175,800,3214

Record 5: 60,IRVING,F,STERN,D11,6423,9/14/1973,MANAGER,16,M,7/7/1945,32250,500,2580

Record 6: 70,EVA,D,PULASKI,D21,7831,9/30/1980,MANAGER,16,F,5/26/1953,36170,700,2893

Record 7: 90,EILEEN,W,HENDERSON,E11,5498,8/15/1970,MANAGER,16,F,5/15/1941,29750,600,2380

Record 8: 100,THEODORE,Q,SPENSER,E21,972,6/19/1980,MANAGER,14,M,12/18/1956,26150,500,2092
```

## PUTF procedure - Write a formatted string to a file

The PUTF procedure writes a formatted string to a specified file.

## Syntax



## Procedure parameters

*file* An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

*format*
An input argument of type VARCHAR(1024) the specifies the string to use for formatting the text. The special character sequence, %s, is substituted by the value of *argN*. The special character sequence, \n, indicates a new line.

*argN*
An optional input argument of type VARCHAR(1024) that specifies a value to substitute in the format string for the corresponding occurrence of the special character sequence %s. Up to five arguments, *arg1* through *arg5*, can be specified. *arg1* is substituted for the first occurrence of %s, *arg2* is substituted for the second occurrence of %s, and so on.

## Authorization

EXECUTE privilege on the UTL_FILE module.

## Example

Format employee data.

```
SET SERVEROUTPUT ON@

CREATE PROCEDURE proc1()
BEGIN
  DECLARE v_filehandle    UTL_FILE.FILE_TYPE;
  DECLARE v_dirAlias      VARCHAR(50) DEFAULT 'mydir';
  DECLARE v_filename      VARCHAR(20) DEFAULT 'myfile.csv';
  DECLARE v_format        VARCHAR(200);
  SET v_format = '%s %s, %s\nSalary: $%s Commission: $%s\n\n';
  SET v_filehandle = UTL_FILE.FOPEN(v_dirAlias,v_filename,'w');
  CALL UTL_FILE.PUTF(v_filehandle,v_format,'000030','SALLY','KWAN','40175','3214');
  CALL DBMS_OUTPUT.PUT_LINE('Wrote to file: ' || v_filename);
  CALL UTL_FILE.FCLOSE(v_filehandle);
END@

CALL proc1@
```

This example results in the following output:

```
Wrote formatted text to file: myfile.csv
```

The formatted file, `myfile.csv`, contains the following data:

```
000030 SALLY, KWAN
Salary: $40175 Commission: $3214
```

# UTL_FILE.FILE_TYPE

UTL_FILE.FILE_TYPE is a file handle type that is used by routines in the UTL_FILE module.

### Example

Declare a variable of type UTL_FILE.FILE_TYPE.

```
DECLARE  v_filehandle    UTL_FILE.FILE_TYPE;
```

## UTL_MAIL module

The UTL_MAIL module provides the capability to send e-mail.

The schema for this module is SYSIBMADM.

The UTL_MAIL module includes the following routines.

*Table 33. System-defined routines available in the UTL_MAIL module*

| Routine name | Description |
|---|---|
| SEND procedure | Packages and sends an e-mail to an SMTP server. |
| SEND_ATTACH_RAW procedure | Same as the SEND procedure, but with BLOB attachments. |
| SEND_ATTACH_VARCHAR2 | Same as the SEND procedure, but with VARCHAR attachments |

### Usage notes

In order to successfully send an e-mail using the UTL_MAIL module, the database configuration parameter SMTP_SERVER must contain one or more valid SMTP server addresses.

### Examples

*Example 1:* To set up a single SMTP server with the default port 25:

```
db2 update db cfg using smtp_server 'smtp.ibm.com'
```

*Example 2:* To set up a single SMTP server that uses port 2000, rather than the default port 25:

```
db2 update db cfg using smtp_server 'smtp2.ibm.com:2000'
```

*Example 3:* To set a list of SMTP servers:

```
db2 update db cfg using smtp_server
   'smtp.example.com,smtp1.example.com:23,smtp2.example.com:2000'
```

**Note:** The e-mail is sent to each of the SMTP servers, in the order listed, until a successful reply is received from one of the SMTP servers.

## SEND procedure - Send an e-mail to an SMTP server

The SEND procedure provides the capability to send an e-mail to an SMTP server.

### Syntax

```
►►—SEND—(—sender—,—recipients—,—cc—,—bcc—,—subject—,—message————————►
►─┬──────────────────────────────┬─)──────────────────────────────────►◄
  └─,─mime_type─┬──────────────┬─┘
                └─,─priority───┘
```

## Parameters

*sender*
An input argument of type VARCHAR(256) that specifies the e-mail address of the sender.

*recipients*
An input argument of type VARCHAR(32672) that specifies the comma-separated e-mail addresses of the recipients.

*cc* An input argument of type VARCHAR(32672) that specifies the comma-separated e-mail addresses of copy recipients.

*bcc*
An input argument of type VARCHAR(32672) that specifies the comma-separated e-mail addresses of blind copy recipients.

*subject*
An input argument of type VARCHAR(32672) that specifies the subject line of the e-mail.

*message*
An input argument of type VARCHAR(32672) that specifies the body of the e-mail.

*mime_type*
An optional input argument of type VARCHAR(1024) that specifies the MIME type of the message. The default is `'text/plain; charset=us-ascii'`.

*priority*
An optional argument of type INTEGER that specifies the priority of the e-mail The default value is 3.

## Authorization

EXECUTE privilege on the UTL_MAIL module.

## Examples

*Example 1:* The following anonymous block sends a simple e-mail message.

```
BEGIN
  DECLARE v_sender VARCHAR(30);
  DECLARE v_recipients VARCHAR(60);
  DECLARE v_subj VARCHAR(20);
  DECLARE v_msg VARCHAR(200);

  SET v_sender = 'kkent@mycorp.com';
  SET v_recipients = 'bwayne@mycorp.com,pparker@mycorp.com';
  SET v_subj = 'Holiday Party';
  SET v_msg = 'This year''s party is scheduled for Friday, Dec. 21 at ' ||
     '6:00 PM. Please RSVP by Dec. 15th.';
  CALL UTL_MAIL.SEND(v_sender, v_recipients, NULL, NULL, v_subj, v_msg);
END@
```

This example results in the following output:

```
BEGIN
  DECLARE v_sender VARCHAR(30);
  DECLARE v_recipients VARCHAR(60);
  DECLARE v_subj VARCHAR(20);
  DECLARE v_msg VARCHAR(200);

  SET v_sender = 'kkent@mycorp.com';
  SET v_recipients = 'bwayne@mycorp.com,pparker@mycorp.com';
  SET v_subj = 'Holiday Party';
  SET v_msg = 'This year''s party is scheduled for Friday, Dec. 21 at ' ||
     '6:00 PM. Please RSVP by Dec. 15th.';
  CALL UTL_MAIL.SEND(v_sender, v_recipients, NULL, NULL, v_subj, v_msg);
END
DB20000I  The SQL command completed successfully.
```

## SEND_ATTACH_RAW procedure - Send an e-mail with a BLOB attachment to an SMTP server

The SEND_ATTACH_RAW procedure provides the capability to send an e-mail to an SMTP server with a binary attachment.

### Syntax

```
►►──SEND_ATTACH_RAW──(──sender──,──recipients──,──cc──,──bcc──,──subject──,──────────►

►─message──,──mime_type──,──priority──,──attachment──────────────────────────────────►

►──┬──────────────────────────────────────────────────────────────┬──)──────────────►◄
   └─,──att_inline──┬───────────────────────────────────────────┬─┘
                    └─,──att_mime_type──┬────────────────────┬──┘
                                        └─,──att_filename────┘
```

### Parameters

*sender*
An input argument of type VARCHAR(256) that specifies the e-mail address of the sender.

*recipients*
An input argument of type VARCHAR(32672) that specifies the comma-separated e-mail addresses of the recipients.

*cc* An input argument of type VARCHAR(32672) that specifies the comma-separated e-mail addresses of copy recipients.

*bcc*
An input argument of type VARCHAR(32672) that specifies the comma-separated e-mail addresses of blind copy recipients.

*subject*
An input argument of type VARCHAR(32672) that specifies the subject line of the e-mail.

*message*
An input argument of type VARCHAR(32672) that specifies the body of the e-mail.

*mime_type*
An input argument of type VARCHAR(1024) that specifies the MIME type of the message. The default is `'text/plain; charset=us-ascii'`.

*priority*
An input argument of type INTEGER that specifies the priority of the e-mail The default value is 3.

*attachment*
An input argument of type BLOB(10M) that contains the attachment.

*att_inline*
An optional input argument of type BOOLEAN that specifies whether the attachment is viewable inline. If set to "true", then the attachment is viewable inline, "false" otherwise. The default value is "true".

*att_mime_type*
An optional input argument of type VARCHAR(1024) that specifies the MIME type of the attachment. The default value is `application/octet`.

*att_filename*
An optional input argument of type VARCHAR(512) that specifies the file name containing the attachment. The default value is NULL.

### Authorization

EXECUTE privilege on the UTL_MAIL module.

## SEND_ATTACH_VARCHAR2 procedure - Send an e-mail with a VARCHAR attachment to an SMTP server

The SEND_ATTACH_VARCHAR2 procedure provides the capability to send an e-mail to an SMTP server with a text attachment.

### Syntax

```
►►─SEND_ATTACH_VARCHAR2─(─sender─,─recipients─,─cc─,─bcc─,─subject─,──────►

►─message─,─mime_type─,─priority─,─attachment─────────────────────────────►

►─┬────────────────────────────────────────────────┬─)──────────────────►◄
  └─,─att_inline─┬───────────────────────────────┬─┘
                 └─,─att_mime_type─┬───────────┬─┘
                                   └─,─att_filename─┘
```

### Parameters

*sender*
An input argument of type VARCHAR(256) that specifies the e-mail address of the sender.

*recipients*
An input argument of type VARCHAR(32672) that specifies the comma-separated e-mail addresses of the recipients.

*cc* An input argument of type VARCHAR(32672) that specifies the comma-separated e-mail addresses of copy recipients.

*bcc*
An input argument of type VARCHAR(32672) that specifies the comma-separated e-mail addresses of blind copy recipients.

*subject*
An input argument of type VARCHAR(32672) that specifies the subject line of the e-mail.

*message*
An input argument of type VARCHAR(32672) that specifies the body of the e-mail.

*mime_type*
An input argument of type VARCHAR(1024) that specifies the MIME type of the message. The default is `'text/plain; charset=us-ascii'`.

*priority*
An input argument of type INTEGER that specifies the priority of the e-mail The default value is 3.

*attachment*
An input argument of type VARCHAR(32000) that contains the attachment.

*att_inline*
An optional input argument of type BOOLEAN that specifies whether the attachment is viewable inline. If set to "true", then the attachment is viewable inline, "false" otherwise. The default value is "true".

*att_mime_type*
An optional input argument of type VARCHAR(1024) that specifies the MIME type of the attachment. The default value is `'text/plain; charset=us-ascii'`.

*att_filename*
An optional input argument of type VARCHAR(512) that specifies the file name containing the attachment. The default value is NULL.

### Authorization

EXECUTE privilege on the UTL_MAIL module.

## UTL_SMTP module

The UTL_SMTP module provides the capability to send e-mail over the Simple Mail Transfer Protocol (SMTP).

The UTL_SMTP module includes the following routines.

*Table 34. System-defined routines available in the UTL_SMTP module*

| Routine Name | Description |
|---|---|
| CLOSE_DATA procedure | Ends an e-mail message. |
| COMMAND procedure | Execute an SMTP command. |
| COMMAND_REPLIES procedure | Execute an SMTP command where multiple reply lines are expected. |
| DATA procedure | Specify the body of an e-mail message. |
| EHLO procedure | Perform initial handshaking with an SMTP server and return extended information. |
| HELO procedure | Perform initial handshaking with an SMTP server. |
| HELP procedure | Send the HELP command. |
| MAIL procedure | Start a mail transaction. |
| NOOP procedure | Send the null command. |
| OPEN_CONNECTION function | Open a connection. |
| OPEN_CONNECTION procedure | Open a connection. |

Table 34. System-defined routines available in the UTL_SMTP module (continued)

| Routine Name | Description |
|---|---|
| OPEN_DATA procedure | Send the DATA command. |
| QUIT procedure | Terminate the SMTP session and disconnect. |
| RCPT procedure | Specify the recipient of an e-mail message. |
| RSET procedure | Terminate the current mail transaction. |
| VRFY procedure | Validate an e-mail address. |
| WRITE_DATA procedure | Write a portion of the e-mail message. |
| WRITE_RAW_DATA procedure | Write a portion of the e-mail message consisting of RAW data. |

The following table lists the public variables available in the module.

Table 35. System-defined types available in the UTL_SMTP module

| Public variable | Data type | Description |
|---|---|---|
| connection | RECORD | Description of an SMTP connection. |
| reply | RECORD | SMTP reply line. |

The CONNECTION record type provides a description of an SMTP connection.

```
ALTER MODULE SYSIBMADM.UTL_SMTP PUBLISH TYPE connection AS ROW
(
  /* name or IP address of the remote host running SMTP server */
    host VARCHAR(255),
  /* SMTP server port number */
    port INTEGER,
  /* transfer timeout in seconds */
    tx_timeout INTEGER,
);
```

The REPLY record type provides a description of an SMTP reply line. REPLIES is an array of SMTP reply lines.

```
ALTER MODULE SYSIBMADM.UTL_SMTP PUBLISH TYPE reply AS ROW
(
  /* 3 digit reply code received from the SMTP server */
    code INTEGER,
  /* the text of the message received from the SMTP server */
    text VARCHAR(508)
);
```

## Examples

*Example 1:* The following procedure constructs and sends a text e-mail message using the UTL_SMTP module.

```
CREATE OR REPLACE PROCEDURE send_mail(
IN p_sender VARCHAR(4096),
IN p_recipient VARCHAR(4096),
IN p_subj VARCHAR(4096),
IN p_msg VARCHAR(4096),
IN p_mailhost VARCHAR(4096))
SPECIFIC send_mail
LANGUAGE SQL
BEGIN
  DECLARE v_conn UTL_SMTP.CONNECTION;
  DECLARE v_crlf VARCHAR(2);
  DECLARE v_port INTEGER CONSTANT 25;

  SET v_crlf = CHR(13) || CHR(10);
  SET v_conn = UTL_SMTP.OPEN_CONNECTION(p_mailhost, v_port, 10);
  CALL UTL_SMTP.HELO(v_conn, p_mailhost);
  CALL UTL_SMTP.MAIL(v_conn, p_sender);
  CALL UTL_SMTP.RCPT(v_conn, p_recipient);
  CALL UTL_SMTP.DATA(
    v_conn,
    'Date: ' || TO_CHAR(SYSDATE, 'Dy, DD Mon YYYY HH24:MI:SS') || v_crlf ||
    'From: ' || p_sender || v_crlf ||
    'To: ' || p_recipient || v_crlf ||
    'Subject: ' || p_subj || v_crlf ||
    p_msg);
  CALL UTL_SMTP.QUIT(v_conn);
END@

CALL send_mail('bwayne@mycorp.com','pparker@mycorp.com','Holiday Party',
'Are you planning to attend?','smtp.mycorp.com')@
```

*Example 2:* The following example uses the OPEN_DATA, WRITE_DATA, and CLOSE_DATA procedures instead of the DATA procedure.

```
CREATE OR REPLACE PROCEDURE send_mail_2(
IN p_sender VARCHAR(4096),
IN p_recipient VARCHAR(4096),
IN p_subj VARCHAR(4096),
IN p_msg VARCHAR(4096),
IN p_mailhost VARCHAR(4096)) SPECIFIC send_mail_2
LANGUAGE SQL
BEGIN
  DECLARE v_conn UTL_SMTP.CONNECTION;
  DECLARE v_crlf VARCHAR(2);
  DECLARE v_port INTEGER CONSTANT 25;

  SET v_crlf = CHR(13) || CHR(10);
  SET v_conn = UTL_SMTP.OPEN_CONNECTION(p_mailhost, v_port, 10);
  CALL UTL_SMTP.HELO(v_conn, p_mailhost);
  CALL UTL_SMTP.MAIL(v_conn, p_sender);
  CALL UTL_SMTP.RCPT(v_conn, p_recipient);
  CALL UTL_SMTP.OPEN_DATA(v_conn);
  CALL UTL_SMTP.WRITE_DATA(v_conn, 'From: ' || p_sender || v_crlf);
  CALL UTL_SMTP.WRITE_DATA(v_conn, 'To: ' || p_recipient || v_crlf);
  CALL UTL_SMTP.WRITE_DATA(v_conn, 'Subject: ' || p_subj || v_crlf);
  CALL UTL_SMTP.WRITE_DATA(v_conn, v_crlf || p_msg);
  CALL UTL_SMTP.CLOSE_DATA(v_conn);
  CALL UTL_SMTP.QUIT(v_conn);
END@

CALL send_mail_2('bwayne@mycorp.com','pparker@mycorp.com','Holiday Party',
'Are you planning to attend?','smtp.mycorp.com')@
```

## CLOSE_DATA procedure - End an e-mail message

The CLOSE_DATA procedure terminates an e-mail message

The procedure terminates an e-mail message by sending the following sequence:

```
<CR><LF>.<CR><LF>
```

This is a single period at the beginning of a line.

### Syntax

```
►►──CLOSE_DATA──(──c──┬─────────┬──)─────────────────────────────────►◄
                      └─,─reply─┘
```

### Parameters

*c* An input or output argument of type CONNECTION that specifies the SMTP connection to be closed.

*reply*
An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## COMMAND procedure - Run an SMTP command

The COMMAND procedure provides the capability to execute an SMTP command.

**Note:** Use COMMAND_REPLIES if multiple reply lines are expected to be returned.

### Syntax

```
►►──COMMAND──(──c──,──cmd──,──┬─────────────────┬──)──────────────────►◄
                              └─arg──,──reply───┘
```

### Parameters

*c* An input or output argument of type CONNECTION that specifies the SMTP connection to which the command is to be sent.

*cmd*
An input argument of type VARCHAR(510) that specifies the SMTP command to process.

*arg*
An optional input argument of type VARCHAR(32672) that specifies an argument to the SMTP command. The default is NULL.

*reply*
An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## COMMAND_REPLIES procedure - Run an SMTP command where multiple reply lines are expected

The COMMAND_REPLIES function processes an SMTP command that returns multiple reply lines.

**Note:** Use COMMAND if only a single reply line is expected.

### Syntax

```
►►──COMMAND_REPLIES──(──c──,──cmd──,──┬────────────────────┬──)──────────────────►◄
                                      └─arg──,──replies─┘
```

### Parameters

*c*
An input or output argument of type CONNECTION that specifies the SMTP connection to which the command is to be sent.

*cmd*
An input argument of type VARCHAR(510) that specifies the SMTP command to process.

*arg*
An optional input argument of type VARCHAR(32672) that specifies an argument to the SMTP command. The default is NULL.

*replies*
An optional output argument of type REPLIES that returns multiple reply lines from the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## DATA procedure - Specify the body of an e-mail message

The DATA procedure provides the capability to specify the body of the e-mail message.

The message is terminated with a `<CR><LF>.<CR><LF>` sequence.

### Syntax

```
►►──DATA──(──c──,──body──┬──────────┬──)─────────────────────────────────────►◄
                         └─,──reply─┘
```

### Parameters

*c*
An input or output argument of type CONNECTION that specifies the SMTP connection to which the command is to be sent.

*body*
An input argument of type VARCHAR(32000) that specifies the body of the e-mail message to be sent.

*reply*
An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## EHLO procedure - Perform initial handshaking with an SMTP server and return extended information

The EHLO procedure performs initial handshaking with the SMTP server after establishing the connection.

The EHLO procedure allows the client to identify itself to the SMTP server. The HELO procedure performs the equivalent functionality, but returns less information about the server.

### Syntax

```
►►──EHLO──(──c──,──domain──┬─────────────┬──)──────────────────────────►◄
                           └─,──replies──┘
```

### Parameters

*c*
An input or output argument of type CONNECTION that specifies the connection to the SMTP server over which to perform handshaking.

*domain*
An input argument of type VARCHAR(255) that specifies the domain name of the sending host.

*replies*
An optional output argument of type REPLIES that return multiple reply lines from the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## HELO procedure - Perform initial handshaking with an SMTP server

The HELO procedure performs initial handshaking with the SMTP server after establishing the connection.

The HELO procedure allows the client to identify itself to the SMTP server. The EHLO procedure performs the equivalent functionality, but returns more information about the server.

### Syntax

```
>>──HELO──(──c──,──domain──┬─────────────┬──)─────────────────────────────────────><
                           └─,──reply────┘
```

### Parameters

*c*
An input or output argument of type CONNECTION that specifies the connection to the SMTP server over which to perform handshaking.

*domain*
An input argument of type VARCHAR(255) that specifies the domain name of the sending host.

*reply*
An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## HELP procedure - Send the HELP command

The HELP function provides the capability to send the HELP command to the SMTP server.

### Syntax

```
>>──HELP──(──c──┬────────────────────────┬──)─────────────────────────────────────><
                └─command──,──replies────┘
```

### Parameters

*c*
An input or output argument of type CONNECTION that specifies the SMTP connection to which the command is to be sent.

*command*
: An optional input argument of type VARCHAR(510) that specifies the command about which help is requested.

*replies*
: An optional output argument of type REPLIES that returns multiple reply lines from the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## MAIL procedure - Start a mail transaction

### Syntax

```
▸▸──MAIL──(──c──,──sender──┬───────────────────────────────┬──)──────────────────────▸◂
                           └─,──parameters──,──reply─┘
```

### Parameters

*c* An input or output argument of type CONNECTION that specifies the connection to the SMTP server on which to start a mail transaction.

*sender*
: An input argument of type VARCHAR(256) that specifies the e-mail address of the sender.

*parameters*
: An optional input argument of type VARCHAR(32672) that specifies the optional mail command parameters in the format `key=value`.

*reply*
: An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## NOOP procedure - Send the null command

The NOOP procedure sends a null command to the SMTP server. The NOOP has no effect on the server except to obtain a successful response.

### Syntax

```
►►──NOOP──(──c──┬─────────────┬──)──────────────────────────────────────►◄
                └──,──reply───┘
```

### Parameters

*c* An input or output argument of type CONNECTION that specifies the SMTP connection on which to send the command.

*reply*
An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## OPEN_CONNECTION function - Return a connection handle to an SMTP server

The OPEN_CONNECTION function returns a connection handle to an SMTP server.

The function returns a connection handle to the SMTP server.

### Syntax

```
►►──OPEN_CONNECTION──(──host──,──port──,──tx_timeout──)────────────────►◄
```

### Parameters

*host*
An input argument of type VARCHAR(255) that specifies the name of the SMTP server.

*port*
An input argument of type INTEGER that specifies the port number on which the SMTP server is listening.

*tx_timeout*
An input argument of type INTEGER that specifies the time out value, in seconds. To instruct the procedure not to wait, set this value to 0. To instruct the procedure to wait indefinitely, set this value to NULL.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

## OPEN_CONNECTION procedure - Open a connection to an SMTP server

The OPEN_CONNECTION procedure opens a connection to an SMTP server.

### Syntax

```
►►─OPEN_CONNECTION─(─host─,─port─,─connection─,─tx_timeout─,─reply─)──────►◄
```

### Parameters

*host*
: An input argument of type VARCHAR(255) that specifies the name of the SMTP server.

*port*
: An input argument of type INTEGER that specifies the port number on which the SMTP server is listening.

*connection*
: An output argument of type CONNECTION that returns a connection handle to the SMTP server.

*tx_timeout*
: An optional input argument of type INTEGER that specifies the time out value, in seconds. To instruct the procedure not to wait, set this value to 0. To instruct the procedure to wait indefinitely, set this value to NULL.

*reply*
: An output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

## OPEN_DATA procedure - Send the DATA command to the SMTP server

The OPEN_DATA procedure sends the DATA command to the SMTP server.

### Syntax

```
►►─OPEN_DATA─(─c─┬──────────┬─)──────────────────────────────────────────►◄
                 └─,─reply──┘
```

### Parameters

*c*
: An input argument of type CONNECTION that specifies the SMTP connection on which to send the command

*reply*
: An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## QUIT procedure - Close the session with the SMTP server

The QUIT procedure closes the session with an SMTP server.

### Syntax

```
►►──QUIT──(──c──┬────────────┬──)──────────────────────────────────────────►◄
                └─,──reply───┘
```

### Parameters

*c* An input or output argument of type CONNECTION that specifies the SMTP connection to terminate.

*reply*
An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## RCPT procedure - Provide the e-mail address of the recipient

The RCPT procedure provides the e-mail address of the recipient.

**Note:** To schedule multiple recipients, invoke the RCPT procedure multiple times.

### Syntax

```
►►──RCPT──(──c──,──recipient──┬───────────────────────────┬──)────────────►◄
                              └─,──parameters──,──reply───┘
```

### Parameters

*c* An input or output argument of type CONNECTION that specifies the SMTP connection on which to add a recipient.

*recipient*
An input argument of type VARCHAR(256) that specifies the e-mail address of the recipient.

*parameters*
An optional input argument of type VARCHAR(32672) that specifies the mail command parameters in the format `key=value`.

*reply*
An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## RSET procedure - End the current mail transaction

The RSET procedure provides the capability to terminate the current mail transaction.

### Syntax

```
►►—RSET—(—c—┬───────────┬—)——————————————————————————————————►◄
            └—,—reply—┘
```

### Parameters

*c* An input or output argument of type CONNECTION that specifies the SMTP connection on which to cancel the mail transaction.

*reply*
An optional output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## VRFY procedure - Validate and verify the recipient's e-mail address

The VRFY function provides the capability to validate and verify a recipient e-mail address. If valid, the recipient’s full name and fully qualified mailbox is returned.

### Syntax

```
►►—VRFY—(—c—,—recipient—,—reply—)————————————————————————►◄
```

### Parameters

*c*
An input or output argument of type CONNECTION that specifies the SMTP connection on which to verify the e-mail address.

*recipient*
An input argument of type VARCHAR(256) that specifies the e-mail address to be verified.

*reply*
An output argument of type REPLY that returns a single reply line from the SMTP server. It is the last reply line if multiple reply lines are returned by the SMTP server.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

### Usage notes

This procedure can be invoked using function invocation syntax in a PL/SQL assignment statement.

## WRITE_DATA procedure - Write a portion of an e-mail message

The WRITE_DATA procedure provides the capability to add data to an e-mail message. The WRITE_DATA procedure may be called repeatedly to add data.

### Syntax

```
►►──WRITE_DATA──(──c──,──data──)──────────────────────────────────────────►◄
```

### Parameters

*c*
An input or output argument of type CONNECTION that specifies the SMTP connection on which to add data.

*data*
An input argument of type VARCHAR(32000) that specifies the data to be added to the e-mail message.

### Authorization

EXECUTE privilege on the UTL_SMTP module.

## WRITE_RAW_DATA procedure - Add RAW data to an e-mail message

The WRITE_RAW_DATA procedure provides the capability to add data to an e-mail message. The WRITE_RAW_DATA procedure may be called repeatedly to add data.

### Syntax

```
►►──WRITE_RAW_DATA──(──c──,──data──)──────────────────────────────────────►◄
```

## Parameters

*c*
An input or output argument of type CONNECTION that specifies the SMTP connection on which to add data.

*data*
An input argument of type BLOB(15M) that specifies the data to be added to the e-mail message.

## Authorization

EXECUTE privilege on the UTL_SMTP module.

# Chapter 4. DB2 compatibility features

## Introduction to DB2 compatibility features

DB2 Version 9.5 introduced a number of features that greatly simplify the task of enabling some applications written for different relational database products to run on DB2. DB2 Version 9.7 introduces additional features that reduce this complexity and the time required to enable existing applications even further.

Some of these features, including the following, are enabled by default.

- Implicit casting (weak typing), which reduces the amount of SQL that needs to be modified when applications that currently run on other data servers are enabled for DB2
- New scalar functions. For details, see "Supported functions and administrative SQL routines and views".
- Major improvements to the TIMESTAMP_FORMAT and VARCHAR_FORMAT scalar functions. (TO_DATE and TO_TIMESTAMP are synonyms for TIMESTAMP_FORMAT, and TO_CHAR is a synonym for VARCHAR_FORMAT.)
  - TIMESTAMP_FORMAT – This function returns a timestamp that is based on the interpretation of the input string using the specified format.
  - VARCHAR_FORMAT – This function returns a string representation of an input expression that has been formatted according to a specified character template.
- The lifting of several SQL language restrictions, resulting in compatible syntax between products; for example, the use of correlation names in subqueries and table functions is now optional
- Synonyms for syntax used by other database products; for example:
  - UNIQUE is a synonym for DISTINCT in the context of column functions and the select list of a query
  - MINUS is a synonym for the EXCEPT set operator
  - *seqname*.NEXTVAL and *seqname*.CURRVAL can be used in place of the SQL standard syntax NEXT VALUE FOR *seqname* and PREVIOUS VALUE FOR *seqname*
- Global variables, which can be used to easily map package variables, emulate @@nested, @@level, or @errorlevel global variables, or pass information from DB2 applications down to triggers, functions, or procedures
- An ARRAY collection data type that can be used to easily map to VARRAY constructs in SQL procedures
- Increased identifier length limits that facilitate the enablement of applications from other DBMS vendors on DB2
- The pseudocolumn ROWID that can be used to refer to the RID; an unqualified ROWID reference is equivalent to RID_BIT() and a qualified ROWID, such as EMPLOYEE.ROWID, is equivalent to RID_BIT(EMPLOYEE)

Other features can be selectively enabled by setting a new DB2 registry variable named **DB2_COMPATIBILITY_VECTOR**. These features are disabled by default.

- An implementation of hierarchical queries using CONNECT BY PRIOR syntax
- Support for outer joins using the outer join operator, (+)
- Use of the DATE data type as TIMESTAMP(0), a combined date and time value

© Copyright IBM Corp. 1993, 2010

- Syntax and semantics to support the NUMBER data type
- Syntax and semantics to support the VARCHAR2 data type
- A pseudocolumn named ROWNUM is a synonym for ROW_NUMBER() OVER(), but ROWNUM is allowed in the SELECT LIST and in the WHERE clause
- A dummy table named DUAL provides a capability that is similar to SYSIBM.SYSDUMMY1
- Alternate semantics for the TRUNCATE TABLE statement, under which IMMEDIATE is an optional keyword that is assumed to be the default if not specified. An implicit commit operation is performed before the TRUNCATE statement executes if the TRUNCATE statement is not the first statement in the logical unit of work.
- Support for assignment of the CHAR or GRAPHIC data type (instead of the VARCHAR or VARGRAPHIC data type) to character and graphic string constants whose byte length is less than or equal to 254
- Use of collection methods to perform operations on arrays, such as first, last, next, and previous
- Support for the creation of Oracle data dictionary-compatible views
- Support for the compilation and execution of PL/SQL statements and language elements
- Support for making cursors insensitive to subsequent statements by materializing the cursor at OPEN time.
- Support for INOUT parameters in procedures that are defined with defaults and can be invoked without specifying the arguments for those parameters.

### Additional resources

For more information, see DB2 Viper 2 compatibility features.

For information about the IBM Migration Toolkit (MTK), see Migrate Now!.

For information on the DB2 Oracle database compatibility features, see Oracle to DB2 Conversion Guide: Compatibility Made Easy.

## DB2_COMPATIBILITY_VECTOR registry variable

The **DB2_COMPATIBILITY_VECTOR** registry variable is used to enable one or more DB2 compatibility features introduced since DB2 Version 9.5.

These features ease the task of migrating applications written for other relational database vendors to DB2 Version 9.5 or later.

**Important:** Enable these features only if they are required for a specific compatibility purpose. If DB2 compatibility features are enabled, some SQL behavior is changed from what is documented in the SQL reference information. To determine the potential impacts on your SQL applications, see the documentation that is associated with each compatibility setting.

This DB2 registry variable is represented as a hexadecimal value, and each bit in the variable enables one of the DB2 compatibility features.

- Operating systems: All

- Default: NULL; Values: NULL or 00 to 3FFF. To take full advantage of these DB2 compatibility features, set the value to ORA for Oracle applications and SYB for Sybase applications. These are the recommended settings.

## Registry variable settings

*Table 36.* **DB2_COMPATIBILITY_VECTOR** *values*

| Bit position | Compatibility feature | Description |
|---|---|---|
| 1 (0x01) | ROWNUM | Enables the use of ROWNUM as a synonym for ROW_NUMBER() OVER(), and permits ROWNUM to appear in the WHERE clause of SQL statements. |
| 2 (0x02) | DUAL | Resolves unqualified table references to 'DUAL' as SYSIBM.DUAL. |
| 3 (0x04) | Outer join operator | Enables support for the outer join operator (+). |
| 4 (0x08) | Hierarchical queries | Enables support for hierarchical queries using the CONNECT BY clause. |
| 5 (0x10) | NUMBER data type [1] | Enables the NUMBER data type and associated numeric processing. |
| 6 (0x20) | VARCHAR2 data type [1] | Enables the VARCHAR2 and NVARCHAR2 data types and associated character string processing. |
| 7 (0x40) | DATE data type [1] | Enables use of the DATE data type as TIMESTAMP(0), a combined date and time value. |
| 8 (0x80) | TRUNCATE TABLE | Enables alternate semantics for the TRUNCATE statement, under which IMMEDIATE is an optional keyword that is assumed to be the default if not specified. An implicit commit operation is performed before the TRUNCATE statement executes if the TRUNCATE statement is not the first statement in the logical unit of work. |
| 9 (0x100) | Character literals | Enables the assignment of the CHAR or GRAPHIC data type (instead of the VARCHAR or VARGRAPHIC data type) to character and graphic string constants whose byte length is less than or equal to 254. |
| 10 (0x200) | Collection methods | Enables the use of methods to perform operations on arrays, such as first, last, next, and previous. Also enables the use of parentheses in place of square brackets in references to specific elements in an array; for example, array1(*i*) refers to element *i* of array1. |
| 11 (0x400) | Data dictionary-compatible views [1] | Enables the creation of data dictionary-compatible views. |
| 12 (0x800) | PL/SQL compilation [2] | Enables the compilation and execution of PL/SQL statements and language elements. |

*Table 36.* **DB2_COMPATIBILITY_VECTOR** *values (continued)*

| Bit position | Compatibility feature | Description |
|---|---|---|
| 13 (0x1000) | Insensitive cursor | Enables cursors defined WITH RETURN to be insensitive if the select-statement does not explicitly specify FOR UPDATE |
| 14 (0x2000) | "INOUT parameter" on page 385 | Enables the specification of DEFAULT for INOUT parameter declarations |
| 1. Applicable only during database creation. Enabling or disabling this feature only affects subsequently created databases.<br>2. See "Restrictions on PL/SQL support". | | |

## Usage

The **DB2_COMPATIBILITY_VECTOR** registry variable is set and updated using the db2set command:

- To enable all of the supported Oracle compatibility features, set the registry variable to the value ORA (equivalent to the hexadecimal value FFF).
- To enable all of the supported Sybase compatibility features, set the registry variable to the value SYB (equivalent to the hexadecimal value 3004).

When the **DB2_COMPATIBILITY_VECTOR** registry variable is set, all databases created should be created as UNICODE databases.

A new setting for the registry variable does not take effect until after the instance has been stopped and then restarted. Existing DB2 packages must be rebound for the change to take effect; packages that are not rebound explicitly will pick up the change on the next implicit rebind.

## Example 1

This example sets the registry variable to enable all of the supported Oracle compatibility features:

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
```

## Example 2

This example shows how to disable all compatibility features by resetting the **DB2_COMPATIBILITY_VECTOR** registry variable:

```
db2set DB2_COMPATIBILITY_VECTOR=
db2stop
db2start
```

Note that if a database has been created with the NUMBER data type or the VARCHAR2 data type enabled, use of the DATE data type as TIMESTAMP(0) enabled, or the creation of Oracle data dictionary-compatible views enabled, the database will still be enabled for these features after this db2set command executes.

## Setting up DB2 for Oracle application enablement

Oracle applications can be enabled to work with DB2 data servers when the DB2 environment is set up appropriately.

- A DB2 data server product must be installed.
- You require the authority to issue the db2set command.
- You require the authority to issue the CREATE DATABASE command.

**Note:** The Control Center tools, which have been deprecated in DB2 Version 9.7 and might be discontinued in a future release, are not supported in an environment that is enabled for the DB2 compatibility features.

Perform this task if you want to enable Oracle applications in DB2 environments. A DB2 environment will support many commonly referenced features from other database vendors. This task is a prerequisite for executing PL/SQL statements or SQL statements that reference Oracle data types from DB2 interfaces, or for any other SQL compatibility features. DB2 compatibility features are enabled at the database level and cannot be disabled.

1. Open a DB2 command window.
2. Start the DB2 database manager.
   ```
   db2start
   ```
3. Set the **DB2_COMPATIBILITY_VECTOR** registry variable to the hexadecimal value that enables the compatibility features that you want to use. To take full advantage of these DB2 compatibility features, set the value to ORA. This is the recommended setting.
   ```
   db2set DB2_COMPATIBILITY_VECTOR=ORA
   ```
4. Set the **DB2_DEFERRED_PREPARE_SEMANTICS** registry variable to YES to enable deferred prepare support. If the **DB2_COMPATIBILITY_VECTOR** registry variable is set to ORA, and the **DB2_DEFERRED_PREPARE_SEMANTICS** registry variable is not set, a default value of YES is used. However, it is recommended that the **DB2_DEFERRED_PREPARE_SEMANTICS** registry variable be explicitly set to YES.
   ```
   db2set DB2_DEFERRED_PREPARE_SEMANTICS=YES
   ```
5. Issue the db2stop command and the db2start command to stop and then restart the database manager.
   ```
   db2stop
   db2start
   ```
6. Create your DB2 database by issuing the CREATE DATABASE command. The database should be created as a UNICODE database, which is the default. For example, to create a database named DB, issue the following command:
   ```
   db2 CREATE DATABASE DB
   ```
7. Optional: Run a CLPPlus or command line processor (CLP) script (`script.sql`) to verify that the database supports PL/SQL statements and data types. The following CLPPlus script creates a simple procedure and then calls that procedure.
   ```
   CONNECT user@hostname:port/dbname;

   CREATE TABLE t1 (c1 NUMBER);

   CREATE OR REPLACE PROCEDURE testdb(num IN NUMBER, message OUT VARCHAR2)
   AS
   BEGIN
    INSERT INTO t1 VALUES (num);
   ```

```
 message := 'The number you passed is: ' || TO_CHAR(num);
END;
/

CALL testdb(100, ?);

DISCONNECT;
EXIT;
```

To run the CLPPlus script, issue the following command:

```
clpplus @script.sql
```

The following example shows the CLP version of the same script. This script uses the SET SQLCOMPAT PLSQL command to enable recognition of the forward slash character (/) on a new line as a PL/SQL statement termination character.

```
CONNECT TO DB;

SET SQLCOMPAT PLSQL;

-- Semicolon is used to terminate
-- the CREATE TABLE statement:
CREATE TABLE t1 (c1 NUMBER);

-- Forward slash on a new line is used to terminate
-- the CREATE PROCEDURE statement:
CREATE OR REPLACE PROCEDURE testdb(num IN NUMBER, message OUT VARCHAR2)
AS
BEGIN
 INSERT INTO t1 VALUES (num);

 message := 'The number you passed is: ' || TO_CHAR(num);
END;
/

CALL testdb(100, ?);

SET SQLCOMPAT DB2;

CONNECT RESET;
```

To run the CLP script, issue the following command:

```
db2 -tvf script.sql
```

The DB2 database that you created is enabled for Oracle applications. The compatibility features that you enabled can now be used.

- Start using the CLPPlus interface.
- Execute PL/SQL scripts and statements.
- Transfer database object definitions.
- Enable database applications.

## Sybase application migration

IBM DB2 SQL Skin Feature 1.0 for applications compatible with Sybase ASE (IBM DB2 SSacSA) helps you migrate your Sybase Adaptive Server Enterprise (ASE) applications to run against DB2 databases. This migration is accomplished with little or no change to your existing source code.

With IBM DB2 SSacSA, your Sybase application behaves as if it was still accessing a Sybase database, even though the new target is a DB2 database. Your tried-and-true application code runs unchanged on DB2 stored data. In most cases all that is required is resetting your connection parameters so that your application connects to IBM DB2 SSacSA instead of your Sybase server. Compared to a manual migration, IBM DB2 SSacSA saves you significant time and effort. Because little or no change is required for your existing applications, you can experience the following benefits:

- You will require fewer resources, because developers are not needed to rewrite code.
- You can reduce test time, because fewer code changes means less opportunity to introduce new bugs.
- You can focus on the database, because this is where most of the migration work occurs.

In addition, you can use the IBM Migration Toolkit to migrate your database schema and data from your source Sybase databases to DB2 databases. When all your information is stored in the same database, migrated Sybase applications and native DB2 applications can share DB2 instances, native triggers and procedures, and DB2 enhanced security.

### Additional resources

For IBM DB2 SSacSA documentation, see the Sybase application migration guides.

For information about the IBM Migration Toolkit (MTK), see Migrate Now!

# Data types

## DATE data type based on TIMESTAMP(0)

The DATE data type is changed to support applications that use the Oracle DATE data type expecting that the values include time information (for example, '2009-04-01-09.43.05').

Support for DATE to mean TIMESTAMP(0) is at the database level, and must be enabled before creating the database where support is required. This is achieved by setting the **DB2_COMPATIBILITY_VECTOR** registry variable to the appropriate value. When a database is created with DATE as TIMESTAMP(0) enabled, the database configuration parameter **date_compat** is set to ON. After a database is created with DATE as TIMESTAMP(0) support enabled, it cannot be disabled for that database, even if the **DB2_COMPATIBILITY_VECTOR** registry variable is reset. Similarly, all databases created with DATE as TIMESTAMP(0) support disabled cannot have this support enabled, even by setting the **DB2_COMPATIBILITY_VECTOR** registry variable.

### Enablement

DATE as TIMESTAMP(0) support is enabled by setting bit position number 7 (0x40) of the **DB2_COMPATIBILITY_VECTOR** registry variable before creating a database. A new setting for the registry variable does not take effect until after the instance has been stopped and then restarted.

## Usage

The following support is enabled for a DB2 database that has the **date_compat** database configuration parameter set to ON.

When the DATE data type is explicitly encountered in SQL statements, it is implicitly mapped to TIMESTAMP(0). An exception is the specification of SQL DATE in the *xml-index-specification* clause of a CREATE INDEX statement. As a result of this implicit mapping, messages refer to the TIMESTAMP data type instead of DATE, and any operations that describe data types for columns or routines return TIMESTAMP instead of DATE.

Datetime literal support is unchanged in a DB2 database that has the **date_compat** database configuration parameter set to ON, except in the following cases:

- The value of an explicit DATE literal returns a TIMESTAMP(0) value in which the time portion is all zeros. For example, `DATE '2008-04-28'` actually represents the timestamp value '2008-04-28-00.00.00'.
- The database manager supports additional formats for the string representation of a date, which correspond to 'DD-MON-YYYY' and 'DD-MON-RR' in English only (see “TIMESTAMP_FORMAT scalar function” for a description of the format elements). For example, '28-APR-2008' or '28-APR-08' can be used as string representations of a date, which actually represents the TIMESTAMP(0) value '2008-04-28-00.00.00'.

The CURRENT_DATE (and CURRENT DATE) special register returns a TIMESTAMP(0) value that is the same as CURRENT_TIMESTAMP(0).

Adding a numeric value to a TIMESTAMP value or subtracting a numeric value from a TIMESTAMP value assumes that the numeric value represents a number of days. The numeric value can have any numeric data type, and any fractional value is considered to be a fractional portion of a day. For example, `TIMESTAMP '2008-03-28 12:00:00' + 1.3` adds 1 day, 7 hours, and 12 minutes to the TIMESTAMP value, resulting in '2008-03-29 19:12:00'. If you are using expressions for partial days, such as 1/24 (1 hour) or 1/24/60 (1 minute), ensure that the **number_compat** database configuration parameter is set to ON so that the division is performed using DECFLOAT arithmetic.

The results of some functions are changed under **date_compat** mode:

- The ADD_MONTHS scalar function with a string argument returns TIMESTAMP(0).
- The DATE scalar function returns TIMESTAMP(0) for all input types.
- The LAST_DAY scalar function with a string argument returns TIMESTAMP(0).
- Other scalar functions that returned DATE based on a DATE input argument (ADD_MONTHS, LAST_DAY, NEXT_DAY, ROUND, and TRUNCATE) return TIMESTAMP, because DATE is always a TIMESTAMP(0) under **date_compat** mode.
- Addition of date values returns TIMESTAMP(0), because date values are really TIMESTAMP(0).
- Subtraction of timestamp values returns DECFLOAT(34), representing the difference as a number of days. Similarly, subtraction of date values returns DECFLOAT(34), which also represents a number of days, because date values are really TIMESTAMP(0).

For databases that have DATE as TIMESTAMP(0) support enabled, if you use the import or load utility to input data into a DATE column (which is based on TIMESTAMP(0)), you must use the TIMESTAMPFORMAT modifier instead of the DATEFORMAT modifier.

## NUMBER data type

The NUMBER data type is introduced to support applications that use the Oracle NUMBER data type.

Support for NUMBER is at the database level, and must be enabled before creating the database where support is required. This is achieved by setting the **DB2_COMPATIBILITY_VECTOR** registry variable to the appropriate value. When a database is created with NUMBER enabled, the database configuration parameter **number_compat** is set to ON. After a database is created with NUMBER support enabled, it cannot be disabled for that database, even if the **DB2_COMPATIBILITY_VECTOR** registry variable is reset. Similarly, all databases created without NUMBER support enabled cannot have NUMBER support enabled, even by setting the **DB2_COMPATIBILITY_VECTOR** registry variable.

### Enablement

NUMBER data type support is enabled by setting bit position number 5 (0x10) of the **DB2_COMPATIBILITY_VECTOR** registry variable before creating a database. To take full advantage of these DB2 compatibility features, set the value to ORA. This is the recommended setting. A new setting for the registry variable does not take effect until after the instance has been stopped and then restarted.

### Usage

The following support is enabled for a DB2 database that has the **number_compat** database configuration parameter set to ON.

When the NUMBER data type is explicitly encountered in SQL statements, it is implicitly mapped as follows:

- If NUMBER is specified without precision and scale attributes, it is mapped to DECFLOAT(16).
- If NUMBER(*p*) is specified, it is mapped to DECIMAL(*p*)
- If NUMBER(*p*,*s*) is specified, it is mapped to DECIMAL(*p*,*s*)

The maximum supported precision is 31, and the scale must be a positive value no greater than the precision. As a result of this implicit mapping, messages will refer to data types DECFLOAT and DECIMAL instead of NUMBER, and any operations that describe data types for columns or routines will return either DECIMAL or DECFLOAT instead of NUMBER. Note that DECFLOAT(16) provides a lower maximum precision than the Oracle NUMBER data type. If more than 16 digits of precision are needed for storing numbers in tables, the columns with this requirement should be defined explicitly as DECFLOAT(34).

Numeric literal support is unchanged in a DB2 database that has the **number_compat** configuration parameter set to ON. The rules for integer, decimal, and floating-point constants continue to apply. This limits decimal literals to 31 digits and floating-point literals to the range of binary double-precision floating-point values. A string to DECFLOAT(34) cast (using the CAST specification or the DECFLOAT function) can be used for values beyond the ranges of DECIMAL or DOUBLE up to the range of DECFLOAT(34).

When NUMBER data values are cast to character strings (using either the CAST specification or the VARCHAR or CHAR scalar function), all leading zeros are stripped from the result.

There is currently no support for a numeric literal that ends in either D or F, representing 64-bit binary floating-point and 32-bit binary floating-point values, respectively. A numeric literal that includes an E has the data type of DOUBLE and can be cast to REAL using the CAST specification or the cast function REAL.

In a DB2 database that has the **number_compat** configuration parameter set to ON, the default data type that is used for a sequence value in the CREATE SEQUENCE statement is DECIMAL(27) instead of INTEGER.

In a DB2 database that has the **number_compat** configuration parameter set to ON, all arithmetic operations and arithmetic or mathematical functions involving DECIMAL or DECFLOAT data types are effectively performed using decimal floating-point arithmetic and return a value with a data type of DECFLOAT(34). This also applies to arithmetic operations where both operands have DECIMAL or DECFLOAT(16) data types, which differs from the description of decimal arithmetic in the DB2 SQL Reference. (See "Expressions with arithmetic operators" in "Expressions".) Additionally, all division operations involving only integer data types (SMALLINT, INTEGER, or BIGINT) are effectively performed using decimal floating-point arithmetic and return a value with a data type of DECFLOAT(34) instead of an integer data type (division by zero with integer operands returns infinity and a warning instead of an error).

Function resolution is also changed, such that an argument of data type DECIMAL is considered to be a DECFLOAT value during the resolution process. For purposes of function resolution, this effectively treats functions with arguments that correspond to the NUMBER(*p*[,*s*]) data type as if the argument data type were NUMBER.

This change in function resolution does not apply to the set of functions that have a variable number of arguments and base their result data type on the set of data types of the arguments. The functions included in this set are:

- COALESCE
- DECODE
- GREATEST
- LEAST
- MAX (scalar)
- MIN (scalar)
- NVL
- VALUE

When the **number_compat** configuration parameter is set to ON, the rules for result data types are extended to make DECFLOAT(34) the result data type if the precision of a DECIMAL result data type would have exceeded 31. These rules also apply to corresponding columns in set operations (UNION, EXCEPT(MINUS), and INTERSECT), expression values in the IN list of an IN predicate, and corresponding expressions of a multiple row VALUES clause.

The rounding mode used for assignments and casts that happen on the database server depends on the data types that are involved. In some cases, truncation is used. In cases where the target is a binary floating-point (REAL or DOUBLE) value, round-half-even is used, as usual. In other cases (usually involving a DECIMAL or DECFLOAT value), the rounding is based on the value of the **decflt_rounding** database configuration parameter. This parameter defaults to round-half-even, but can be set to round-half-up to match the Oracle rounding mode. The following table summarizes the rounding that is used for various numeric assignments and casts.

*Table 37. Rounding for numeric assignments and casts*

| Source Data Type | Target Data Type | | | |
|---|---|---|---|---|
| | **integer types** | **DECIMAL** | **DECFLOAT** | **REAL/DOUBLE** |
| **integer types** | not applicable | not applicable | decflt_rounding | round_half_even |
| **DECIMAL** | decflt_rounding | decflt_rounding | decflt_rounding | round_half_even |
| **DECFLOAT** | decflt_rounding | decflt_rounding | decflt_rounding | round_half_even |
| **REAL/DOUBLE** | truncate | truncate | decflt_rounding | round_half_even |
| **string (cast only)** | not applicable | decflt_rounding | decflt_rounding | round_half_even |

The DB2 decimal floating-point values are based on the IEEE 754R standard. Retrieval of DECFLOAT data and casting of DECFLOAT data to character strings removes any trailing zeros after the decimal point.

### Client-server compatibility considerations

- Client applications working with a DB2 database server that is enabled for NUMBER data type support never see a NUMBER data type from the server. Any column or expression that might have reported NUMBER from an Oracle server will report either DECIMAL or DECFLOAT from a DB2 server.
- Because an Oracle environment expects the rounding mode to be round-half-up, it is important that the client rounding mode match the server rounding mode. This means that the `db2cli.ini` setting should match the value of the **decflt_rounding** database configuration parameter. ROUND_HALF_UP should be specified to most closely match the Oracle rounding mode.

### Restrictions

NUMBER data type support has the following restrictions:

- There is no support for the NUMBER data type with a precision attribute greater than 31, a precision attribute of asterisk (*), a scale attribute that exceeds the precision attribute, or a negative scale attribute. There is no corresponding DECIMAL precision and scale support for such data type specifications.
- The trigonometric functions and the DIGITS scalar function cannot be invoked with arguments of data type NUMBER without a precision (DECFLOAT).
- A distinct type cannot be created with the name NUMBER.

## VARCHAR2 and NVARCHAR2 data types

The VARCHAR2 and NVARCHAR2 data types are introduced to support applications that use the Oracle VARCHAR2 and NVARCHAR2 data type.

Support for VARCHAR2 and NVARCHAR2 (subsequently jointly referred to as VARCHAR2) is at the database level, and must be enabled before creating the database where support is required. This is achieved by setting the **DB2_COMPATIBILITY_VECTOR** registry variable to the appropriate value. When a database is created with VARCHAR2 support enabled, the database configuration parameter **varchar2_compat** is set to ON. After a database is created with VARCHAR2 support enabled, it cannot be disabled for that database, even if the **DB2_COMPATIBILITY_VECTOR** registry variable is reset. Similarly, all databases created with VARCHAR2 support disabled cannot have VARCHAR2 support enabled, even by setting the **DB2_COMPATIBILITY_VECTOR** registry variable.

## Enablement

VARCHAR2 data type support is enabled by setting bit position number 6 (0x20) of the **DB2_COMPATIBILITY_VECTOR** registry variable before creating a database. A new setting for the registry variable does not take effect until after the instance has been stopped and then restarted.

To make use of the NVARCHAR2 data type, the database must be a Unicode database.

## Usage

The following support is enabled for a DB2 database that has the **varchar2_compat** database configuration parameter set to ON.

When the VARCHAR2 data type is explicitly encountered in SQL statements, it is implicitly mapped to the VARCHAR data type. The maximum length for VARCHAR2 is the same as the maximum length for VARCHAR (that is, 32 672).

When the NVARCHAR2 data type is explicitly encountered in SQL statements, it is implicitly mapped to the VARGRAPHIC data type. The maximum length for NVARCHAR2 is the same as the maximum length for VARGRAPHIC (that is, 16 336).

Character string literals up to 254 bytes in length have a data type of CHAR. Character string literals longer than 254 bytes have a data type of VARCHAR.

Any comparisons involving varying-length string types use non-padded comparison semantics, and comparisons with only fixed-length string types continue to use blank-padded comparison semantics, with two exceptions:

- Comparisons involving any string column information from catalog views always use the IDENTITY collation with blank-padded comparison semantics, regardless of the database collation.
- String comparisons involving a data type with the FOR BIT DATA attribute always use the IDENTITY collation with blank-padded comparison semantics.

If the result type for the IN list of an IN predicate would resolve to a fixed-length string data type and the left operand of the IN predicate is a varying-length string data type, the IN list expressions are treated as having a varying-length string data type.

Character string values (other than LOB values) with a length of zero are generally treated as null values. An assignment or cast of an empty string value to CHAR, NCHAR, VARCHAR, or NVARCHAR produces a null value.

The functions that return character string arguments, or that are based on parameters with character string data types, also treat empty string CHAR, NCHAR, VARCHAR, or NVARCHAR values as null values. Special considerations apply for some functions when the **varchar2_compat** database configuration parameter is set to ON, and these are listed here.

- CONCAT and the concatenation operator. A null or empty string value is ignored in the concatenated result. The result type of the concatenation is shown in the following table.

*Table 38. Data Type and lengths of concatenated operands*

| Operands | Combined length attributes | Result |
|---|---|---|
| CHAR(A) CHAR(B) | <255 | CHAR(A+B) |
| CHAR(A) CHAR(B) | >254 | VARCHAR(A+B) |
| CHAR(A) VARCHAR(B) | - | VARCHAR(MIN(A+B,32672)) |
| VARCHAR(A) VARCHAR(B) | - | VARCHAR(MIN(A+B,32672)) |
| CLOB(A) CHAR(B) | - | CLOB(MIN(A+B, 2G)) |
| CLOB(A) VARCHAR(B) | - | CLOB(MIN(A+B, 2G)) |
| CLOB(A) CLOB(B) | | CLOB(MIN(A+B, 2G)) |
| GRAPHIC(A) GRAPHIC(B) | <128 | GRAPHIC(A+B) |
| GRAPHIC(A) GRAPHIC(B) | >128 | VARGRAPHIC(A+B) |
| GRAPHIC(A) VARGRAPHIC(B) | - | VARGRAPHIC(MIN(A+B,16336)) |
| VARGRAPHIC(A) VARGRAPHIC(B) | - | VARGRAPHIC(MIN(A+B,16336)) |
| DBCLOB(A) CHAR(B) | - | DBCLOB(MIN(A+B, 1G)) |
| DBCLOB(A) VARCHAR(B) | - | DBCLOB(MIN(A+B, 1G)) |
| DBCLOB(A) CLOB(B) | | DBCLOB(MIN(A+B, 1G)) |
| BLOB(A) BLOB(B) | - | BLOB(MIN(A+B, 2G)) |

- INSERT. A null value or empty string as the fourth argument results in deletion of the number of bytes indicated by the third argument, beginning at the byte position indicated by the second argument from the first argument.
- LENGTH. The value returned by the LENGTH function is the number of bytes in the character string. An empty string value returns the null value.
- REPLACE. If all of the argument values have a data type of CHAR, VARCHAR, GRAPHIC, or VARGRAPHIC, then:
  - A null value or empty string as the second argument is treated as an empty string, and consequently the first argument is returned as the result
  - A null value or empty string as the third argument is treated as an empty string, and nothing replaces the string that is removed from the source string by the second argument.

  If any argument value has a data type of CLOB or BLOB and any argument is the null value, the result is the null value. All three arguments of the REPLACE function must be specified.
- SUBSTR: References to SUBSTR which have a character string input for the first argument will be replaced with an invocation to SUBSTRB. References to SUBSTR which have a graphic string input for the first argument are left unchanged. In this case, there is no support for second argument values less than 1, or third argument values less than zero.
- TRANSLATE. The *from-string-exp* is the second argument, and the *to-string-exp* is the third argument. If the *to-string-exp* is shorter than the *from-string-exp*, the extra characters in the *from-string-exp* that are found in the *char-string-exp* (the

first argument) are removed; that is, the default *pad-char* argument is effectively an empty string, unless a different pad character is specified in the fourth argument.

- TRIM. If the trim character argument of a TRIM function invocation is a null value or an empty string, the function returns a null value.

In the ALTER TABLE statement or the CREATE TABLE statement, when a DEFAULT clause is specified without an explicit value for a column defined with the VARCHAR or the VARGRAPHIC data type, the default value is a blank character.

Empty strings are converted to a blank character when the database configuration parameter **varchar2_compat** is set to ON. For example:

- SYSCAT.DATAPARTITIONS.STATUS has a single blank character when the data partition is visible.
- SYSCAT.PACKAGES.PKGVERSION has a single blank character when the package version has not been explicitly set.
- SYSCAT.ROUTINES.COMPILE_OPTIONS has a null value when compile options have not been set.

**Note:** If SQL statements use parameter markers, a data type conversion that affects VARCHAR2 usage can occur. For example, if the input value is a VARCHAR of length zero and it is converted to a LOB, the result will be a null value. However, if the input value is a LOB of length zero and it is converted to a LOB, the result will be a LOB of length zero. The data type of the input value can be affected by deferred prepare.

### Restrictions

The VARCHAR2 data type and associated character string processing support have the following restrictions:

- The VARCHAR2 length attribute qualifier CHAR is not accepted.
- The LONG VARCHAR and LONG VARGRAPHIC data types are not supported (but are not explicitly blocked) when the **varchar2_compat** database configuration parameter is set to ON.

## Character and graphic constant handling

An alternate way to parse character or graphic constants is introduced to support applications that expect these constants to be assigned the data types CHAR and GRAPHIC, respectively.

### Enablement

This support is enabled by setting bit position number 9 (0x100) of the **DB2_COMPATIBILITY_VECTOR** registry variable. The recommended setting for DB2_COMPATIBILITY_VECTOR is ORA, which sets all of the compatibility bits.

After this support is enabled, character or graphic string constants whose length is less than or equal to 254 bytes have a data type of CHAR or GRAPHIC, respectively. Character or graphic string constants whose length is greater than 254 bytes have a data type of VARCHAR or VARGRAPHIC, respectively. Because this data type assignment can change the result types of some SQL statements, it is strongly recommended that this registry variable setting for a database not be toggled.

## Outer join operator

When the **DB2_COMPATIBILITY_VECTOR** registry variable is set to support the outer join operator, (+), queries can use this alternative syntax within predicates of the WHERE clause.

The outer join syntax should be used wherever possible, and the outer join operator should be used only when enabling applications from other relational database vendors on DB2.

The outer join operator, (+), is applied following a column name within predicates that generally refer to columns from two table-references.

- To write a query that performs a left outer join of tables T1 and T2, include both tables in the FROM clause separated by a comma, and apply the outer join operator to all columns of T2 in predicates that also reference T1. For example:

  ```
  SELECT * FROM T1
    LEFT OUTER JOIN T2 ON T1.PK1 = T2.FK1
      AND T1.PK2 = T2.FK2
  ```

  is equivalent to:

  ```
  SELECT * FROM T1, T2
    WHERE T1.PK1 = T2.FK1(+)
      AND T1.PK2 = T2.FK2(+)
  ```

- To write a query that performs a right outer join of tables T1 and T2, include both tables in the FROM clause separated by a comma, and apply the outer join operator to all columns of T1 in predicates that also reference T2. For example:

  ```
  SELECT * FROM T1
    RIGHT OUTER JOIN T2 ON T1.FK1 = T2.PK1
      AND T1.FK2 = T2.PK2
  ```

  is equivalent to:

  ```
  SELECT * FROM T1, T2
    WHERE T1.FK1(+) = T2.PK1
      AND T1.FK2(+) = T2.PK2
  ```

A table-reference that has columns marked with the outer join operator is sometimes referred to as a *NULL-producer*.

A set of predicates separated by AND operators is known as an *AND-factor*. If there are no AND operators in a WHERE clause, the set of predicates in the WHERE clause is considered to be the only AND-factor.

The following rules apply to the use of the outer join operator.

- The WHERE predicate is considered on a granularity of ANDed Boolean factors.
- Each Boolean term can refer to at most two table-references; that is, `T1.C11 + T2.C21 = T3.C3(+)` is not allowed.
- Each table can be the null producer with respect to at most one other table. If a table is joined to a third table, it must be the outer.
- Local predicates, such as `T1.A(+) = 5`, can exist, but they are executed with the join. A local predicate without (+) is executed after the join.
- Correlation for outer join Boolean terms is not allowed.

- The outer join operator cannot be specified in the same subselect as the explicit JOIN syntax.
- The outer join operator can be specified only in the WHERE clause on columns associated with table-references that are specified in the FROM clause of the same subselect.
- An AND-factor can have only one table-reference as a NULL-producer. Each column reference followed by the outer join operator must be from the same table-reference.
- An AND-factor that includes an outer join operator can reference at most two table-references.
- If multiple AND-factors are required for the outer join between two tables, the outer join operator must be specified in all of these AND-factors. If an AND-factor does not specify the outer join operator, it is processed on the result of the outer join.
- The outer join operator cannot be applied to an entire expression. Within an AND-factor, each column reference from the same table must be followed by the outer join operator (for example, `T1.COL1 (+) - T1.COL2 (+) = T2.COL1`).
- An AND-factor with predicates that involve only one table-reference can specify the outer join operator if there is at least one other AND-factor that involves the same table-reference as the NULL-producer and involves another table-reference as the outer table.
- An AND-factor with predicates involving only one table-reference and without an outer join operator is processed on the result of the join.
- A table-reference can be used only once as the NULL-producer for one other table-reference within a query.
- The same table-reference cannot be used as both the outer table and the NULL-producer in separate outer joins that form a cycle. A cycle can be formed across multiple joins when the chain of predicates comes back to an earlier table-reference. For example:

  ```
  SELECT ... FROM T1,T2,T3
    WHERE T1.a1 = T2.b2(+)
      AND T2.b2 = T3.c3(+)
      AND T3.c3 = T1.a1(+)   -- invalid cycle
  ```

  This example starts with T1 as the outer table in the first predicate and then cycles back to T1 in the third predicate. Note that T2 is used as both the NULL-producer in the first predicate and the outer table in the second predicate, but this usage is not itself a cycle.
- An AND-factor that includes an outer join operator must follow the rules for a join-condition of an ON clause defined under joined-table.
- The outer join operator can only be specified in the WHERE clause on columns that are associated with table-references specified in the FROM clause of the same subselect.

## Hierarchical queries

Hierarchical queries are a form of recursive query that provides support for retrieving a hierarchy, such as a bill of materials, from relational data using a CONNECT BY clause.

Hierarchical query support is enabled through the setting of the **DB2_COMPATIBILITY_VECTOR** registry variable. This allows the CONNECT BY syntax to be specified, including the use of pseudocolumns (see “Pseudocolumns” on page 375 on page 375), unary operators (see “Unary operators”), and the SYS_CONNECT_BY_PATH scalar function.

Connect-by recursion uses the same subquery for the seed (start) and the recursive step (connect). This combination provides a concise method of representing recursions such as, for example, bills-of-material, reports-to-chains, or email threads.

Connect-by recursion returns an error if a cycle occurs. A *cycle* occurs when a row produces itself, either directly or indirectly. Using the optional CONNECT BY NOCYCLE clause, the recursion can be directed to ignore the duplicated row, thus avoiding both the cycle and the error. For more information about mapping hierarchical queries to DB2 recursion, see Port CONNECT BY to DB2.

### Pseudocolumns

A *pseudocolumn* is a qualified or unqualified identifier that has meaning in a specific context and shares the same namespace as columns and variables. If an unqualified identifier does not identify a column or a variable, it is checked to see if it identifies a pseudocolumn.

LEVEL is a pseudocolumn for use in hierarchical queries. The LEVEL pseudocolumn returns the recursive step in the hierarchy at which a row was produced. All rows produced by the START WITH clause return the value 1. Rows produced by applying the first iteration of the CONNECT BY clause return 2, and so on. The data type of the column is INTEGER NOT NULL.

LEVEL must be specified in the context of a hierarchical query but cannot be specified in the START WITH clause, as an argument of the CONNECT_BY_ROOT operator, or as an argument of the SYS_CONNECT_BY_PATH function (SQLSTATE 428H4).

### Unary operators

Unary operators in support of hierarchical queries include:
- CONNECT_BY_ROOT
- PRIOR

### Functions

New functions in support of hierarchical queries include:
- SYS_CONNECT_BY_PATH scalar function

### Subselect

Hierarchical query support includes the following extensions to the subselect.
- The subselect includes a new “hierarchical-query-clause” on page 376
- The clauses of the subselect are processed in the following sequence:
  1. FROM clause
  2. *hierarchical-query-clause*
  3. WHERE clause
  4. GROUP BY clause
  5. HAVING clause

6. SELECT clause
7. ORDER BY clause
8. FETCH FIRST clause

- If the subselect includes a *hierarchical-query-clause*, special rules apply for the order of processing the predicates in the WHERE clause. The *search-condition* is factored into predicates along its AND conditions (conjunction). If a predicate is an implicit join predicate (that is, it references more than one table in the FROM clause), the predicate is applied before the *hierarchical-query-clause*. Any predicate referencing at most one table in the FROM clause is applied to the intermediate result table of the *hierarchical-query-clause*.

  A hierarchical query involving joins should be written using explicit joined tables with an ON clause to avoid confusion about the application of WHERE clause predicates.
- The new ORDER SIBLINGS BY clause can be specified if the subselect includes a *hierarchical-query-clause*. This clause specifies that the ordering applies only to siblings within the hierarchies.

## hierarchical-query-clause

```
►►──●─┬──────────────────────┬─●┤ connect-by-clause ├─●──────────►◄
      └─┤ start-with-clause ├─┘
```

### start-with-clause:

```
├──START WITH──search-condition──────────────────────────────────┤
```

### connect-by-clause:

```
├──CONNECT BY──┬─────────┬──search-condition─────────────────────┤
               └─NOCYCLE─┘
```

A subselect that includes a *hierarchical-query-clause* is called a hierarchical query. After establishing a first intermediate result table $H_1$, subsequent intermediate result tables $H_2$, $H_3$, and so forth are generated by joining $H_n$ with R using the *connect-by-clause* as a join condition to produce $H_{n+1}$. R is the result of the FROM clause of the subselect and any join predicates in the WHERE clause. The process stops when $H_{n+1}$ has yielded an empty result table. The result table H of the *hierarchical-query-clause* is the UNION ALL of every $H_i$.

The *start-with-clause* specifies the intermediate result table $H_1$ for the hierarchical query that consists of those rows of R for which the *search-condition* is true. If the *start-with-clause* is not specified, $H_1$ is the entire intermediate result table R.

The *connect-by-clause* produces the intermediate result table $H_{n+1}$ from $H_n$ by joining $H_n$ with R, using the search condition.

The unary operator PRIOR is used to distinguish column references to $H_n$, the last prior recursive step, from column references to R. For example:

```
CONNECT BY MGRID = PRIOR EMPID
```

MGRID is resolved with R, and EMPID is resolved within the previous intermediate result table $H_n$.

The rules for the *search-condition* within the *start-with-clause* and the *connect-by-clause* are the same as those within the WHERE clause, except that OLAP specifications cannot be specified in the *connect-by-clause* (SQLSTATE 42903).

If the intermediate result table $H_{n+1}$ would return a row from R for a hierarchical path that is the same as a row from R that is already in that hierarchical path, an error is returned (SQLSTATE 560CO).

If the NOCYCLE keyword is specified, an error is not returned, but the repeated row is not included in the intermediate result table $H_{n+1}$.

DB2 supports a maximum of 64 levels of recursion (SQLSTATE 54066).

A subselect that is a hierarchical query returns the intermediate result set in a partial order, unless that order is destroyed through the use of an explicit ORDER BY clause, a GROUP BY or HAVING clause, or a DISTINCT keyword in the select list. The partial order returns rows, such that rows produced in $H_{n+1}$ for a given hierarchy immediately follow the row in $H_n$ that produced them. The ORDER SIBLINGS BY clause can be used to enforce order within a set of rows produced by the same parent.

## Restrictions on the use of hierarchical queries

- A hierarchical query is not supported in a materialized query table (SQLSTATE 428EC).
- The CONNECT BY clause cannot be used in conjunction with XML functions or XQuery (SQLSTATE 428H4).
- A NEXT VALUE expression for a sequence cannot be specified in (SQLSTATE 428F9):
  - The parameter list of the CONNECT_BY_ROOT operator or a SYS_CONNECT_BY_PATH function
  - START WITH and CONNECT BY clauses

## Examples

- The following reports-to-chain example illustrates connect-by recursion. The example is based on a table named MY_EMP, which is created and populated with data as follows:

```
CREATE TABLE MY_EMP(
  EMPID  INTEGER NOT NULL PRIMARY KEY,
  NAME   VARCHAR(10),
  SALARY DECIMAL(9, 2),
  MGRID  INTEGER);

INSERT INTO MY_EMP VALUES ( 1, 'Jones',    30000, 10);
INSERT INTO MY_EMP VALUES ( 2, 'Hall',     35000, 10);
INSERT INTO MY_EMP VALUES ( 3, 'Kim',      40000, 10);
INSERT INTO MY_EMP VALUES ( 4, 'Lindsay',  38000, 10);
INSERT INTO MY_EMP VALUES ( 5, 'McKeough', 42000, 11);
INSERT INTO MY_EMP VALUES ( 6, 'Barnes',   41000, 11);
INSERT INTO MY_EMP VALUES ( 7, 'O''Neil',  36000, 12);
INSERT INTO MY_EMP VALUES ( 8, 'Smith',    34000, 12);
INSERT INTO MY_EMP VALUES ( 9, 'Shoeman',  33000, 12);
INSERT INTO MY_EMP VALUES (10, 'Monroe',   50000, 15);
INSERT INTO MY_EMP VALUES (11, 'Zander',   52000, 16);
INSERT INTO MY_EMP VALUES (12, 'Henry',    51000, 16);
INSERT INTO MY_EMP VALUES (13, 'Aaron',    54000, 15);
INSERT INTO MY_EMP VALUES (14, 'Scott',    53000, 16);
INSERT INTO MY_EMP VALUES (15, 'Mills',    70000, 17);
INSERT INTO MY_EMP VALUES (16, 'Goyal',    80000, 17);
INSERT INTO MY_EMP VALUES (17, 'Urbassek', 95000, NULL);
```

The following query returns all employees working for Goyal, as well as some additional information, such as the reports-to-chain:

```
1 SELECT NAME,
2        LEVEL,
3        SALARY,
4        CONNECT_BY_ROOT NAME AS ROOT,
5        SUBSTR(SYS_CONNECT_BY_PATH(NAME, ':'), 1, 25) AS CHAIN
6   FROM MY_EMP
7   START WITH NAME = 'Goyal'
8   CONNECT BY PRIOR EMPID = MGRID
9   ORDER SIBLINGS BY SALARY;
```

```
NAME       LEVEL       SALARY      ROOT  CHAIN
---------- ----------- ----------- ----- ---------------
Goyal                1    80000.00 Goyal :Goyal
Henry                2    51000.00 Goyal :Goyal:Henry
Shoeman              3    33000.00 Goyal :Goyal:Henry:Shoeman
Smith                3    34000.00 Goyal :Goyal:Henry:Smith
O'Neil               3    36000.00 Goyal :Goyal:Henry:O'Neil
Zander               2    52000.00 Goyal :Goyal:Zander
Barnes               3    41000.00 Goyal :Goyal:Zander:Barnes
McKeough             3    42000.00 Goyal :Goyal:Zander:McKeough
Scott                2    53000.00 Goyal :Goyal:Scott
```

Lines 7 and 8 comprise the core of the recursion: The optional START WITH clause describes the WHERE clause that is to be used on the source table to seed the recursion. In this case, only the row for employee Goyal is selected. If the START WITH clause is omitted, the entire source table is used to seed the recursion. The CONNECT BY clause describes how, given the existing rows, the next set of rows is to be found. The unary operator PRIOR is used to distinguish values in the previous step from those in the current step. PRIOR identifies EMPID as the employee ID of the previous recursive step, and MGRID as originating from the current recursive step.

LEVEL in line 2 is a pseudocolumn that describes the current level of recursion.

CONNECT_BY_ROOT is a unary operator that always returns the value of its argument as it was during the first recursive step; that is, the values that are returned by an explicit or implicit START WITH clause.

SYS_CONNECT_BY_PATH() is a binary function that prepends the second argument to the first and then appends the result to the value that it produced in the previous recursive step. The arguments must be character types.

Unless explicitly overridden, connect-by recursion returns a result set in a partial order; that is, the rows that are produced by a recursive step always follow the row that produced them. Siblings at the same level of recursion have no specific order. The ORDER SIBLINGS BY clause in line 9 defines an order for these siblings, which further refines the partial order, potentially into a total order.

- Return the organizational structure of the DEPARTMENT table. Use the level of the department to visualize the hierarchy.

  ```
  SELECT LEVEL, CAST(SPACE((LEVEL - 1) * 4) || '/' || DEPTNAME
      AS VARCHAR(40)) AS DEPTNAME
    FROM DEPARTMENT
    START WITH DEPTNO = 'A00'
    CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
  ```

  The query returns:

  ```
  LEVEL       DEPTNAME
  ----------- ----------------------------------------
            1 /SPIFFY COMPUTER SERVICE DIV.
  ```

```
2     /PLANNING
2     /INFORMATION CENTER
2     /DEVELOPMENT CENTER
3         /MANUFACTURING SYSTEMS
3         /ADMINISTRATION SYSTEMS
2     /SUPPORT SERVICES
3         /OPERATIONS
3         /SOFTWARE SUPPORT
3         /BRANCH OFFICE F2
3         /BRANCH OFFICE G2
3         /BRANCH OFFICE H2
3         /BRANCH OFFICE I2
3         /BRANCH OFFICE J2
```

## CONNECT_BY_ROOT unary operator

The CONNECT_BY_ROOT unary operator is for use only in hierarchical queries (SQLSTATE 428H4). For every row in the hierarchy, this operator returns the expression for the row's root ancestor.

```
►►──CONNECT_BY_ROOT──expression──────────────────────────────────────►◄
```

*expression*
An expression that does not contain a NEXT VALUE expression, an hierarchical query construct (such as the LEVEL pseudocolumn), the SYS_CONNECT_BY_PATH function, or an OLAP function (SQLSTATE 428H4).

The result type of the operator is the result type of the expression.

Rules:

- A CONNECT_BY_ROOT operator cannot be specified in the START WITH clause or the CONNECT BY clause of a hierarchical query (SQLSTATE 428H4).
- A CONNECT_BY_ROOT operator cannot be specified as an argument to the SYS_CONNECT_BY_PATH function (SQLSTATE 428H4).

Notes:

- A CONNECT_BY_ROOT operator has a higher precedence than any infix operator. Therefore, to pass an expression with infix operators (such as + or ||) as an argument, parentheses must be used. For example:

  ```
  CONNECT_BY_ROOT FIRSTNME || LASTNAME
  ```

  returns the FIRSTNME value of the root ancestor row concatenated with the LASTNAME value of the actual row in the hierarchy, because this expression is equivalent to:

  ```
  (CONNECT_BY_ROOT FIRSTNME) || LASTNAME
  ```

  rather than:

  ```
  CONNECT_BY_ROOT (FIRSTNME || LASTNAME)
  ```

Example:

- Return the hierarchy of departments and their root departments in the DEPARTMENT table.

  ```
  SELECT CONNECT_BY_ROOT DEPTNAME AS ROOT, DEPTNAME
    FROM DEPARTMENT START WITH DEPTNO IN ('B01','C01','D01','E01')
    CONNECT BY PRIOR DEPTNO = ADMRDEPT
  ```

  This query returns:

```
ROOT               DEPTNAME
------------------ -----------------------
PLANNING           PLANNING
INFORMATION CENTER INFORMATION CENTER
DEVELOPMENT CENTER DEVELOPMENT CENTER
DEVELOPMENT CENTER MANUFACTURING SYSTEMS
DEVELOPMENT CENTER ADMINISTRATION SYSTEMS
SUPPORT SERVICES   SUPPORT SERVICES
SUPPORT SERVICES   OPERATIONS
SUPPORT SERVICES   SOFTWARE SUPPORT
SUPPORT SERVICES   BRANCH OFFICE F2
SUPPORT SERVICES   BRANCH OFFICE G2
SUPPORT SERVICES   BRANCH OFFICE H2
SUPPORT SERVICES   BRANCH OFFICE I2
SUPPORT SERVICES   BRANCH OFFICE J2
```

## PRIOR unary operator

The PRIOR unary operator is for use only in the CONNECT BY clause of hierarchical queries (SQLSTATE 428H4).

```
►►──PRIOR──expression────────────────────────────────────────────────►◄
```

The CONNECT BY clause performs an inner join between the intermediate result table $H_n$ of the hierarchical query and the source result table specified in the FROM clause. All column references to tables that are referenced in the FROM clause, and which are arguments to the PRIOR operator, are considered to be ranging over $H_n$.

The primary key of the intermediate result table $H_n$ is typically joined to the foreign keys of the source result table to recursively traverse the hierarchy.

```
CONNECT BY PRIOR T.PK = T.FK
```

If the primary key is a composite key, care must be taken to prefix each column with PRIOR:

```
CONNECT BY PRIOR T.PK1 = T.FK1 AND PRIOR T.PK2 = T.FK2
```

*expression*
Any expression that does not contain a NEXT VALUE expression, an hierarchical query construct (such as the LEVEL pseudocolumn), the SYS_CONNECT_BY_PATH function, or an OLAP function (SQLSTATE 428H4).

The result data type of the operator is the result data type of the expression.

Notes:

- A PRIOR operator has a higher precedence than any infix operator. Therefore, to pass an expression with infix operators (such as + or ||) as an argument, parentheses must be used. For example:

  ```
  PRIOR FIRSTNME || LASTNAME
  ```

  returns the FIRSTNME value of the prior row concatenated with the LASTNAME value of the actual row in the hierarchy, because this expression is equivalent to:

  ```
  (PRIOR FIRSTNME) || LASTNAME
  ```

  rather than:

  ```
  PRIOR (FIRSTNME || LASTNAME)
  ```

Example:

- Return the hierarchy of departments in the DEPARTMENT table.

```
SELECT LEVEL, DEPTNAME
  FROM DEPARTMENT START WITH DEPTNO = 'A00'
  CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
```

This query returns:

```
LEVEL       DEPTNAME
----------- ----------------------------------------
          1 SPIFFY COMPUTER SERVICE DIV.
          2 PLANNING
          2 INFORMATION CENTER
          2 DEVELOPMENT CENTER
          3 MANUFACTURING SYSTEMS
          3 ADMINISTRATION SYSTEMS
          2 SUPPORT SERVICES
          3 OPERATIONS
          3 SOFTWARE SUPPORT
          3 BRANCH OFFICE F2
          3 BRANCH OFFICE G2
          3 BRANCH OFFICE H2
          3 BRANCH OFFICE I2
          3 BRANCH OFFICE J2
```

## SYS_CONNECT_BY_PATH

The SYS_CONNECT_BY_PATH function (in the SYSIBM schema) is used in hierarchical queries to build a string representing a path from the root row to this row.

```
►►──SYS_CONNECT_BY_PATH──(──string-expression1──,──string-expression2──)──────────►◄
```

The string for a given row at LEVEL *n* is built as follows:

- Step 1 (using the values of the root row from the first intermediate result table $H_1$):

  path$_1$ := *string-expression2* || *string-expression1*
- Step *n* (based on the row from the intermediate result table $H_n$):

  path$_n$ := path$_{n-1}$ || *string-expression2* || *string-expression1*

*string-expression1*
: A character string expression that identifies the row. The expression must not include a NEXT VALUE expression for a sequence (SQLSTATE 428F9), any hierarchical query construct, such as the LEVEL pseudocolumn or the CONNECT_BY_ROOT operator (SQLSTATE 428H4), an OLAP function, or an aggregate function (SQLSTATE 428H4).

*string-expression2*
: A constant string that serves as a separator. The expression must not include a NEXT VALUE expression for a sequence (SQLSTATE 428F9), any hierarchical query construct, such as the LEVEL pseudocolumn or the CONNECT_BY_ROOT operator (SQLSTATE 428H4), an OLAP function, or an aggregate function (SQLSTATE 428H4).

The result is a varying-length character string. The length attribute of the result data type is the greater of 1000 and the length attribute of *string-expression1*.

Rules:

- The SYS_CONTEXT_BY_PATH function must not be used outside of the context of a hierarchical query (SQLSTATE 428H4).
- The function cannot be used in a START WITH clause or a CONNECT BY clause (SQLSTATE 428H4).

Example:

- Return the hierarchy of departments in the DEPARTMENT table.

  ```
  SELECT CAST(SYS_CONNECT_BY_PATH(DEPTNAME, '/')
      AS VARCHAR(76)) AS ORG
    FROM DEPARTMENT START WITH DEPTNO = 'A00'
    CONNECT BY NOCYCLE PRIOR DEPTNO = ADMRDEPT
  ```

  This query returns:

  ```
  ORG
  ----------------------------------------------------------------------
  /SPIFFY COMPUTER SERVICE DIV.
  /SPIFFY COMPUTER SERVICE DIV./PLANNING
  /SPIFFY COMPUTER SERVICE DIV./INFORMATION CENTER
  /SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER
  /SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER/MANUFACTURING SYSTEMS
  /SPIFFY COMPUTER SERVICE DIV./DEVELOPMENT CENTER/ADMINISTRATION SYSTEMS
  /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES
  /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/OPERATIONS
  /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/SOFTWARE SUPPORT
  /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE F2
  /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE G2
  /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE H2
  /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE I2
  /SPIFFY COMPUTER SERVICE DIV./SUPPORT SERVICES/BRANCH OFFICE J2
  ```

## Database configuration parameters

New database configuration parameters indicate the status of several compatibility features.

The new parameters include:

- **date_compat**, which indicates whether the DATE compatibility semantics associated with the TIMESTAMP(0) data type are applied to the connected database
- **number_compat**, which indicates whether the compatibility semantics associated with the NUMBER data type are applied to the connected database
- **varchar2_compat**, which indicates whether the compatibility semantics associated with the VARCHAR2 data type are applied to the connected database

The value of each of these parameters is determined at database creation time, and is based on the setting of the **DB2_COMPATIBILITY_VECTOR** registry variable. The value cannot be changed.

## ROWNUM pseudocolumn

DB2 converts any unresolved and unqualified column reference to ROWNUM to the OLAP specification ROW_NUMBER() OVER().

ROWNUM pseudocolumn support is enabled through the setting of the **DB2_COMPATIBILITY_VECTOR** registry variable.

Both ROWNUM and ROW_NUMBER() OVER() are allowed in the WHERE clause of a subselect, and are useful for restricting the size of a result set.

If ROWNUM is used in the WHERE clause, and there is an ORDER BY clause in the same subselect, the ordering is applied before the ROWNUM predicate is evaluated. This is also true for a ROW_NUMBER() OVER() function in the WHERE clause.

If the OLAP specification ROW_NUMBER() OVER() is used in the WHERE clause, neither a window-order-clause nor a window-partition-clause can be specified.

## Notes

- Before translating an unqualified reference to 'ROWNUM' as ROW_NUMBER() OVER(), DB2 attempts to resolve the reference as one of:
  - A column within the current SQL query
  - A local variable
  - A routine parameter
  - A global variable
- Avoid using 'ROWNUM' as a column name or a variable name while ROWNUM pseudocolumn support is enabled.

## Example 1

Set the **DB2_COMPATIBILITY_VECTOR** registry variable to support an application using ROWNUM and outer join operator queries. To achieve maximum compatibility with Oracle, set the value to ORA. This is the recommended setting.

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
```

The new setting does not take effect until after the instance has been stopped and then restarted.

## Example 2

Assuming that ROWNUM pseudocolumn support is enabled for the connected database, retrieve the twentieth to the fortieth row of a result set that is stored in a temporary table.

```
SELECT TEXT FROM SESSION.SEARCHRESULTS
  WHERE ROWNUM BETWEEN 20 AND 40
  ORDER BY ID
```

Note that ROWNUM is affected by the ORDER BY clause.

# DUAL table

The DB2 data server resolves any unqualified table reference to “DUAL” as a built-in view returning one row and one column named “DUMMY”, whose value is 'X'.

Unqualified table references to the DUAL table are resolved as SYSIBM.DUAL through the setting of the *DB2_COMPATIBILITY_VECTOR* registry variable.

If a user-defined table named DUAL exists, the DB2 server resolves a table reference to the user-defined table only if the reference is explicitly qualified.

### Example 1

Generate a random number by selecting from DUAL.

```
SELECT RAND() AS RANDOM_NUMBER FROM DUAL
```

### Example 2

Retrieve the value of the CURRENT SCHEMA special register.

```
SET SCHEMA = MYSCHEMA;
SELECT CURRENT SCHEMA AS CURRENT_SCHEMA FROM DUAL;
```

## Insensitive cursor

Starting with Version 9.7 Fix Pack 2 and later fix packs, you have the option of making cursors insensitive to subsequent statements by materializing the cursor at OPEN time.

For a cursor to be insensitive to other data change statements, the result set is materialized at OPEN time and the cursor behaves as a read only cursor. Without insensitive cursor support there is no guarantee that DB2 cursors will be materialized at OPEN time, which could cause different result sets when a query is run in DB2 as opposed to a relational database that immediately materializes cursors. For example, Sybase TSQL includes the capability of issuing a query from a batch or procedure code that produces a result set for the invoker. The query is materialized immediately and other statements in the block assume that they cannot impact the result and issue statements, such as delete, against the same table that was referenced in the query. When a similar scenario is run without an insensitive cursor, the result set from that cursor will be different than the Sybase result.

Insensitive cursor support is enabled by setting bit position number 13 (0x1000) of the **DB2_COMPATIBILITY_VECTOR** registry variable. When this bit is set, all cursors defined as WITH RETURN are INSENSITIVE as long as they are not explicitly marked as FOR UPDATE. A new setting for the registry variable does not take effect until after the instance has been stopped and then restarted.

The DECLARE CURSOR statement is extended allowing a cursor to be defined as INSENSITIVE. Declaring an insensitive cursor is only supported in the context of a compound SQL (compiled) statement.

The STATICREADONLY option of the BIND command now allows a specification of INSENSITIVE. A package that is bound with STATICREADONLY INSENSITIVE will cause all read only and ambiguous cursors to be insensitive. This bind option will also be supported in the registry variable DB2_SQLROUTINE_PREPOPTS and the procedure SET_ROUTINE_OPTS, so that SQL routines can make all read only and ambiguous cursors issued as static SQL materialize at OPEN time.

### Restrictions

The INSENSITIVE keyword can only be specified on a DECLARE CURSOR statement used within a compound SQL (compiled) statement. It is not supported by any of the precompilers. No changes are made to CLI or JDBC to identify insensitive nonscrollable cursors (either cursor attributes or result set attributes).

### Example

This code returns the entire result set of the SELECT statement to the client prior to starting the DELETE statement.

```
BEGIN
 DECLARE res INSENSITIVE CURSOR WITH RETURN TO CLIENT FOR
  SELECT * FROM T;
  OPEN T;
  DELETE FROM T;
END
```

## INOUT parameter

Starting with Version 9.7 Fix Pack 2 and later fix packs, a procedure can have INOUT parameters defined with a default and the procedure can subsequently be invoked without an argument for those parameters.

Procedure declarations allow the specification of DEFAULT expressions for INOUT parameters in addition to IN parameters. A procedure with INOUT parameters defined with defaults can be invoked without specifying arguments corresponding to those parameters. If an argument corresponding to an INOUT parameter is not specified, or the argument is the DEFAULT keyword, then the provided default expression (or NULL if none was specified) is used to initialize the parameter within the procedure and no value is returned for this parameter when the procedure exits.

INOUT parameter support is enabled by setting bit position number 14 (0x2000) of the **DB2_COMPATIBILITY_VECTOR** registry variable. A new setting for the registry variable does not take effect until after the instance has been stopped and then restarted.

### Restrictions

The DEFAULT keyword is supported for INOUT parameters in procedures, but not in functions.

### Example

Creating a procedure with optional INOUT parameters.

```
CREATE OR REPLACE PROCEDURE paybonus
  (IN empid INTEGER,
  IN percentbonus DECIMAL(2, 2),
  INOUT budget DECFLOAT DEFAULT NULL)
  ...
```

The procedure computes the amount of bonus from the employee's salary, issues the bonus, and then deducts the bonus from the departmental budget. If no budget is given, then that part is ignored. The procedure can be invoked such as:

```
CALL paybonus(12, 0.05, 50000);
CALL paybonus(12, 0.05, DEFAULT);
CALL paybonus(12, 0.05);
```

# Currently committed semantics improve concurrency

Lock timeouts and deadlocks can occur under the CS isolation level with row-level locking, especially with applications that are not designed to prevent such problems. Some high throughput database applications cannot tolerate waiting on locks that are issued during transaction processing, and some applications cannot tolerate processing uncommitted data, but still require non-blocking behavior for read transactions.

Under the new *currently committed* semantics, only committed data is returned, as was the case previously, but now readers do not wait for writers to release row locks. Instead, readers return data that is based on the currently committed version; that is, data prior to the start of the write operation.

Currently committed semantics are turned on by default for new databases. This allows any application to take advantage of the new behavior, and no changes to the application itself are required. The new database configuration parameter **cur_commit** can be used to override this behavior. This might be useful, for example, in the case of applications that require blocking on writers to synchronize internal logic.

Similarly, upgraded databases have **cur_commit** disabled by default in case applications require blocking writers to synchronize their internal logic, and this parameter can be turned on later, if so desired.

Currently committed semantics apply only to read-only scans that do not involve catalog tables or the internal scans that are used to evaluate or enforce constraints. Note that, because currently committed is decided at the scan level, a writer's access plan might include currently committed scans. For example, the scan for a read-only subquery can involve currently committed semantics. Because currently committed semantics obey isolation level semantics, applications running under currently committed semantics continue to respect isolation levels.

Currently committed semantics require increased log space for writers. Additional space is required for logging the first update of a data row during a transaction. This data is required for retrieving the currently committed image of the row. Depending on the workload, this can have an insignificant or measurable impact on the total log space used. The requirement for additional log space does not apply when **cur_commit** is disabled.

## Restrictions

The following restrictions apply to currently committed semantics:

- The target table object in a section that is to be used for data update or deletion operations does not use currently committed semantics. Rows that are to be modified must be lock protected to ensure that they do not change after they have satisfied any query predicates that are part of the update operation.
- A transaction that has made an uncommitted modification to a row forces the currently committed reader to access appropriate log records to determine the currently committed version of the row. Although log records that are no longer in the log buffer can be physically read, currently committed semantics do not support the retrieval of log files from the log archive. This only affects databases that are configured to use infinite logging.
- The following scans do not use currently committed semantics:
  - Catalog table scans

- Scans that are used to enforce referential integrity constraints
- Scans that reference LONG VARCHAR or LONG VARGRAPHIC columns
- Range-clustered table (RCT) scans
- Scans that use spatial or extended indexes

### Example

Consider the following scenario, in which deadlocks are avoided under the currently committed semantics. In this scenario, two applications update two separate tables, but do not yet commit. Each application then attempts to read (with a read-only cursor) from the table that the other application has updated.

| Step | Application A | Application B |
|---|---|---|
| 1 | update T1 set col1 = ? where col2 = ? | update T2 set col1 = ? where col2 = ? |
| 2 | select col1, col3, col4 from T2 where col2 >= ? | select col1, col5, from T1 where col5 = ? and col2 = ? |
| 3 | commit | commit |

Without currently committed semantics, these applications running under the cursor stability isolation level might create a deadlock, causing one of the applications to fail. This happens when each application needs to read data that is being updated by the other application.

Under currently committed semantics, if the query in step 2 (for either application) happens to require the data currently being updated by the other application, that application does not wait for the lock to be released, making a deadlock impossible. The previously committed version of the data is located and used instead.

## Oracle data dictionary-compatible views

When the **DB2_COMPATIBILITY_VECTOR** registry variable is set to support Oracle data dictionary-compatible views, the views are automatically created when the database is created.

Support for Oracle data dictionary-compatible views is at the database level, and must be enabled before creating the database where support is required. The data dictionary definition includes CREATE VIEW, CREATE PUBLIC SYNONYM, and COMMENT statements for each view that is compatible with Oracle's data dictionary. These views, which are created in the SYSIBMADM schema, are listed in Table 39.

*Table 39. Oracle data dictionary-compatible views*

| Category | Defined views |
|---|---|
| General | DICTIONARY, DICT_COLUMNS<br>USER_CATALOG, DBA_CATALOG, ALL_CATALOG<br>USER_DEPENDENCIES, DBA_DEPENDENCIES, ALL_DEPENDENCIES<br>USER_OBJECTS, DBA_OBJECTS, ALL_OBJECTS<br>USER_SEQUENCES, DBA_SEQUENCES, ALL_SEQUENCES<br>USER_TABLESPACES, DBA_TABLESPACES |

*Table 39. Oracle data dictionary-compatible views (continued)*

| Category | Defined views |
|---|---|
| Tables or views | USER_CONSTRAINTS, DBA_CONSTRAINTS, ALL_CONSTRAINTS<br>USER_CONS_COLUMNS, DBA_CONS_COLUMNS, ALL_CONS_COLUMNS<br>USER_INDEXES, DBA_INDEXES, ALL_INDEXES<br>USER_IND_COLUMNS, DBA_IND_COLUMNS, ALL_IND_COLUMNS<br>USER_TAB_PARTITIONS, DBA_TAB_PARTITIONS, ALL_TAB_PARTITIONS<br>USER_PART_TABLES, DBA_PART_TABLES, ALL_PART_TABLES<br>USER_PART_KEY_COLUMNS, DBA_PART_KEY_COLUMNS, ALL_PART_KEY_COLUMNS<br>USER_SYNONYMS, DBA_SYNONYMS, ALL_SYNONYMS<br>USER_TABLES, DBA_TABLES, ALL_TABLES<br>USER_TAB_COMMENTS, DBA_TAB_COMMENTS, ALL_TAB_COMMENTS<br>USER_TAB_COLUMNS, DBA_TAB_COLUMNS, ALL_TAB_COLUMNS<br>USER_COL_COMMENTS, DBA_COL_COMMENTS, ALL_COL_COMMENTS<br>USER_TAB_COL_STATISTICS, DBA_TAB_COL_STATISTICS, ALL_TAB_COL_STATISTICS<br>USER_VIEWS, DBA_VIEWS, ALL_VIEWS<br>USER_VIEW_COLUMNS, DBA_VIEW_COLUMNS, ALL_VIEW_COLUMNS |
| Programming objects | USER_PROCEDURES, DBA_PROCEDURES, ALL_PROCEDURES<br>USER_SOURCE, DBA_SOURCE, ALL_SOURCE<br>USER_TRIGGERS, DBA_TRIGGERS, ALL_TRIGGERS<br>USER_ERRORS, DBA_ERRORS, ALL_ERRORS<br>USER_ARGUMENTS, DBA_ARGUMENTS, ALL_ARGUMENTS |
| Security | USER_ROLE_PRIVS, DBA_ROLE_PRIVS, ROLE_ROLE_PRIVS<br>SESSION_ROLES<br>USER_SYS_PRIVS, DBA_SYS_PRIVS, ROLE_SYS_PRIVS<br>SESSION_PRIVS<br>USER_TAB_PRIVS, DBA_TAB_PRIVS, ALL_TAB_PRIVS, ROLE_TAB_PRIVS<br>USER_TAB_PRIVS_MADE, ALL_TAB_PRIVS_MADE<br>USER_TAB_PRIVS_RECD, ALL_TAB_PRIVS_RECD<br>DBA_ROLES |

## Examples

- Enable the creation of data dictionary-compatible views for a database named MYDB.

```
db2set DB2_COMPATIBILITY_VECTOR=ORA
db2stop
db2start
db2 create db mydb
```

- Determine what data dictionary-compatible views are available.

```
connect to mydb
select * from dictionary
```

- Several data dictionary-compatible views provide information about privileges. Use the USER_SYS_PRIVS view to show all the system privileges that the current user has been granted.

```
connect to mydb
select * from user_sys_privs
```

- Determine the column definitions for the DBA_TABLES view.

```
connect to mydb
describe select * from dba_tables
```

# DB2-Oracle terminology mapping

Because Oracle applications can be enabled to work with DB2 data servers when the DB2 environment is set up appropriately, it is important to understand how certain Oracle concepts map to DB2 concepts.

Table 40 provides a concise summary of commonly used Oracle terms and their DB2 equivalents.

*Table 40. Mapping of common Oracle concepts to DB2 concepts*

| Oracle concept | DB2 concept | Notes |
|---|---|---|
| active log | active log | This is the same concept. |
| actual parameter | argument | This is the same concept. |

*Table 40. Mapping of common Oracle concepts to DB2 concepts (continued)*

| Oracle concept | DB2 concept | Notes |
|---|---|---|
| alert log | db2diag log files and administration notification log | The db2diag log files are primarily intended for use by IBM Software Support for troubleshooting purposes. The administration notification log is primarily intended for troubleshooting use by database and system administrators. Administration notification log messages are also logged to the db2diag log files using a standardized message format. |
| archive log | offline-archive log | This is the same concept. |
| archive log mode | log archiving | This is the same concept. |
| background_dump_dest | diagpath | This is the same concept. |
| created global temporary table | created global temporary table | This is the same concept. |
| cursor sharing | statement concentrator | This is the same concept. |
| data block | data page | This is the same concept. |
| data buffer cache | buffer pool | This is the same concept. However, in DB2 you can have as many buffer pools of any page size you like. |
| data dictionary | system catalog | The DB2 system catalog contains metadata in the form of tables and views. The database manager creates and maintains two sets of system catalog views that are defined on the base system catalog tables:<br>• SYSCAT views, which are read-only views<br>• SYSSTAT views, which are updatable views that contain statistical information that is used by the optimizer |
| data dictionary cache | catalog cache | This is the same concept. |
| data file | container | DB2 data is physically stored in containers, which contain objects. |
| database link | nickname | A nickname is an identifier that refers to an object at a remote data source (a federated database object). |
| dual table | dual table | This is the same concept. |

Table 40. Mapping of common Oracle concepts to DB2 concepts (continued)

| Oracle concept | DB2 concept | Notes |
|---|---|---|
| dynamic performance views | snapshot monitor SQL administrative views | Snapshot monitor SQL administrative views, which use schema SYSIBMADM, return monitor data about a specific area of the database system. For example, the SYSIBMADM.SNAPBP SQL administrative view provides a snapshot of buffer pool information. |
| extent | extent | A DB2 extent is made up of a set of contiguous data pages. |
| formal parameter | parameter | This is the same concept. |
| global index | nonpartitioned index | This is the same concept. |
| inactive log | online-archive log | This is the same concept. |
| init.ora and Server Parameter File (SPFILE) | database manager configuration file and database configuration file | A DB2 instance can contain multiple databases. Therefore, configuration parameters and their values are stored at both the instance level, in the database manager configuration file, and at the database level, in the database configuration file. These files are managed through the GET or UPDATE DBM CFG command and the GET or UPDATE DB CFG command, respectively. |
| instance | instance or database manager | An instance is a combination of background processes and shared memory. A DB2 instance is also known as a database manager. Because a DB2 instance can contain multiple databases, there are DB2 configuration files at both the instance level (the database manager configuration file) and at the database level (the database configuration file). |
| large pool | utility heap | The utility heap is used by the backup, restore, and load utilities. |
| library cache | package cache | The package cache, which is allocated from database shared memory, is used to cache sections for static and dynamic SQL and XQuery statements on a database. |

*Table 40. Mapping of common Oracle concepts to DB2 concepts (continued)*

| Oracle concept | DB2 concept | Notes |
|---|---|---|
| local index | partitioned index | This is the same concept. |
| materialized view | materialized query table (MQT) | An MQT is a table whose definition is based on the results of a query and is meant to be used to improve performance. The DB2 SQL compiler determines whether a query would run more efficiently against an MQT than it would against the base table on which the MQT is based. |
| noarchive log mode | circular logging | This is the same concept. |
| Oracle Call Interface (OCI) | DB2CI Interface | DB2CI is a 'C' and 'C++' application programming interface that uses function calls to connect to DB2 Version 9.7 databases, manage cursors, and perform SQL statements. See "IBM Data Server Driver for DB2CI" on page 395 for a list of OCI APIs supported by the DB2CI driver. |
| Oracle Call Interface (OCI) | Call Level Interface (CLI) | CLI is a C and C++ application programming interface that uses function calls to pass dynamic SQL statements as function arguments. In most cases, you can replace an OCI function with a CLI function and relevant changes to the supporting program code. |
| ORACLE_SID environment variable | DB2INSTANCE environment variable | This is the same concept. |
| partitioned tables | partitioned tables | This is the same concept. |
| Procedural Language/Structured Query Language (PL/SQL) | SQL Procedural Language (SQL PL) | SQL PL is an extension of SQL that consists of statements and language elements. SQL PL provides statements for declaring variables and condition handlers, assigning values to variables, and implementing procedural logic. SQL PL is a subset of the SQL Persistent Stored Modules (SQL/PSM) language standard. Oracle PL/SQL statements can be compiled and executed using DB2 interfaces. |

*Table 40. Mapping of common Oracle concepts to DB2 concepts (continued)*

| **Oracle concept** | **DB2 concept** | **Notes** |
|---|---|---|
| program global area (PGA) | application shared memory and agent private memory | Application shared memory stores information that is shared between a database and a particular application: primarily, rows of data being passed to or from the database. Agent private memory stores information used to service a particular application, such as sort heaps, cursor information, and session contexts. |
| redo log | transaction log | The transaction log records database transactions and can be used for recovery. |
| role | role | This is the same concept. |
| segment | storage object | This is the same concept. |
| session | session; database connection | This is the same concept. |
| startup nomount | db2start | The command that starts the instance. |
| synonym | alias | An alias is an alternative name for a table, view, nickname, or another alias. The term "synonym" is tolerated and can be specified in place of "alias". Aliases are not used to control what version of a DB2 procedure or user-defined function is being used by an application; to do this, use the SET PATH statement to add the required schema to the value of the CURRENT PATH special register. |
| system global area (SGA) | instance shared memory and database shared memory | The instance shared memory stores all of the information for a particular instance, such as lists of all active connections and security information. The database shared memory stores information for a particular database, such as package caches, log buffers, and buffer pools. |
| SYSTEM table space | SYSCATSPACE table space | The SYSCATSPACE table space contains the system catalog. This table space is created by default when you create a database. |
| table space | table space | This is the same concept. |

*Table 40. Mapping of common Oracle concepts to DB2 concepts (continued)*

| Oracle concept | DB2 concept | Notes |
|---|---|---|
| user global area (UGA) | application global memory | Application global memory comprises application shared memory and application-specific memory. |

# Chapter 5. DB2CI application development

DB2CI is a callable SQL interface to the DB2 Version 9.7 database servers. It is a 'C' and 'C++' application programming interface for DB2 database access that uses function calls to connect to databases, manage cursors, and perform SQL statements.

Starting with Version 9.7 Fix Pack 1, you can use the DB2CI interface to access databases on DB2 Version 9.7 servers on any of the supported operating systems.

The DB2CI interface provides support for a number of Oracle Call Interface (OCI) APIs. This support reduces the complexity of enabling existing OCI applications so that they work with DB2 databases. The IBM Data Server Driver for DB2CI is the driver for the DB2CI interface.

## IBM Data Server Driver for DB2CI

The IBM Data Server Driver for DB2CI provides support for DB2CI application development.

The IBM Data Server Client includes the DB2CI driver. You need to install this client to install the DB2CI driver.

The DB2CI driver provides support for calls to the following OCI APIs:

OCIAttrGet

OCIAttrSet

OCIBindArrayOfStruct

OCIBindByName

OCIBindByPos

OCIBindDynamic

OCIBreak

OCIClientVersion

OCIDateAddDays

OCIDateAddMonths

OCIDateAssign

OCIDateCheck

OCIDateCompare

OCIDateDaysBetween

OCIDateFromText

OCIDateLastDay

OCIDateNextDay

OCIDateSysDate

OCIDateToText

OCIDefineArrayOfStruct

OCIDefineByPos

OCILobGetLength

OCILobIsEqual

OCILobIsTemporary

OCILobIsOpen

OCILobLocatorAssign

OCILobLocatorIsInit

OCILobRead

OCILobTrim

OCILobWrite

OCILogoff

OCILogon

OCILogon2

OCINumberAbs

OCINumberAdd

OCINumberArcCos

OCINumberArcSin

OCINumberArcTan

OCINumberArcTan2

OCINumberAssign

OCINumberCeil

OCINumberCmp

OCINumberTan

OCINumberToInt

OCINumberToReal

OCINumberToRealArray

OCINumberToText

OCINumberTrunc

OCIParamGet

OCIParamSet

OCIPasswordChange

OCIPing

OCIRawAllocSize

OCIRawAssignBytes

OCIRawAssignRaw

OCIRawPtr

OCIRawResize

OCIRawSize

OCIReset

OCIResultSetToStmt

OCIServerAttach

OCIServerDetach

OCIServerVersion

© Copyright IBM Corp. 1993, 2010

OCIDefineDynamic
OCIDescribeAny
OCIDescriptorAlloc
OCIDescriptorFree
OCIEnvCreate
OCIEnvInit
OCIErrorGet
OCIFileClose
OCIFileExists
OCIFileFlush
OCIFileGetLength
OCIFileInit
OCIFileOpen
OCIFileRead
OCIFileSeek
OCIFileTerm
OCIFileWrite
OCIHandleAlloc
OCIHandleFree
OCIInitialize
OCILobAppend
OCILobAssign
OCILobClose
OCILobCopy
OCILobCreateTemporary
OCILobDisableBuffering
OCILobEnableBuffering
OCILobErase
OCILobFreeTemporary
OCILobFlushBuffer
OCINumberCos
OCINumberDec
OCINumberDiv
OCINumberExp
OCINumberFloor
OCINumberFromInt
OCINumberFromReal
OCINumberFromText
OCINumberHypCos
OCINumberHypSin
OCINumberHypTan
OCINumberInc
OCINumberIntPower
OCINumberIsInt
OCINumberIsZero
OCINumberLn
OCINumberLog
OCINumberMod
OCINumberMul
OCINumberNeg
OCINumberPower
OCINumberPrec
OCINumberRound
OCINumberSetPi
OCINumberSetZero
OCINumberShift
OCINumberSign
OCINumberSin
OCINumberSqrt
OCINumberSub
OCISessionBegin
OCISessionEnd
OCISessionGet
OCISessionRelease
OCIStmtExecute
OCIStmtFetch
OCIStmtFetch2
OCIStmtGetBindInfo
OCIStmtGetPieceInfo
OCIStmtPrepare
OCIStmtPrepare2
OCIStmtRelease
OCIStmtSetPieceInfo
OCIStringAllocSize
OCIStringAssign
OCIStringAssignText
OCIStringPtr
OCIStringResize
OCIStringSize
OCITerminate
OCITransCommit
OCITransDetach
OCITransForget
OCITransMultiPrepare
OCITransPrepare
OCITransRollback
OCITransStart
xaoEnv
xaosterr
xaoSvcCtx

## Building DB2CI applications

You can build DB2CI applications using an existing Oracle Call Interface (OCI) application and the bldapp script file.

- You must have a DB2 database with the same structure as the Oracle database used by your existing OCI application.
- You must have installed the IBM Data Server Client.

DB2 samples provides a script called `bldapp` for compiling and linking applications that use OCI functions supported by the IBM Data Server Driver for DB2CI. It is located in the *DB2DIR*`\samples\db2ci` or *DB2DIR*`/samples/db2ci` directories, along with sample programs. *DB2DIR* represents the location where your DB2 copy is installed.

The bldapp script file takes up to four parameters. The first parameter, $1, specifies the name of your source file. The additional parameters are only required to build embedded SQL programs that requires a connection to the database: the second parameter, $2, specifies the name of the database to which you want to connect; the third parameter, $3, specifies the user ID for the database, and $4 specifies the password. If the program contains embedded SQL, indicated by the .sqc extension, then the embprep script is called to precompile the program, producing a program file with a .c extension.

**Restriction**

- Ensure that your existing OCI application only has calls to OCI functions supported by the DB2CI driver. See “IBM Data Server Driver for DB2CI” on page 395 for a complete list of supported OCI functions.

1. If you are building your DB2CI application using an existing OCI application, ensure that you specify the db2ci.h include file.
2. Build your DB2CI application with the bldapp script file The following example shows how to build the sample program tbinfo from the source file tbinfo.c on Linux and UNIX operating systems:
   ```
   cd $INSTHOME/sqllib/samples/db2ci
   bldapp tbinfo
   ```
   The result is an executable file, tbinfo.
3. Run the executable file generated in the previous step by entering the executable name as follows:
   ```
   tbinfo
   ```

## DB2CI application compile and link options (AIX)

The compile and link options in this topic are recommended for building DB2CI applications with the AIX® IBM C compiler.

You can find the following options in the *DB2DIR*/samples/cli/bldapp batch file, where *DB2DIR* is the location where your DB2 copy is installed.

Compile and link options for bldapp

**Compile options:**

**xlc** The IBM C compiler.

**$EXTRA_CFLAG**
Contains the value "-q64" for 64-bit environments; otherwise, contains no value.

**-I$DB2PATH/include**
Specify the location of the DB2 include files. For example: $HOME/sqllib/include

**-c** Perform compile only; no link. This script has separate compile and link steps.

**Link options:**

**`xlc`** Use the compiler as a front end for the linker.

**`$EXTRA_CFLAG`**
Contains the value "-q64" for 64-bit environments; otherwise, contains no value.

**`-o $1`** Specify the executable program.

**`$1.o`** Specify the object file.

**`utilci.o`**
Include the utility object file for error checking.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 runtime shared libraries. For example: `$HOME/sqllib/$LIB`. If you do not specify the `-L` option, the compiler assumes the following path: `/usr/lib:/lib`.

**`-ldb2ci`**
Link with the DB2CI library.

## DB2CI application compile and link options (HP-UX)

The compile and link options in this topic are recommended for building DB2CI applications with the HP-UX C compiler.

You can find the following options in the *`DB2DIR`*`/samples/db2ci/bldapp` batch file, where *DB2DIR* is the location where your DB2 copy is installed.

Compile and link options for bldapp

**Compile options:**

**`cc`** Use the C compiler.

**`$EXTRA_CFLAG`**
If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **`+DD64`**; if 32-bit support is enabled, it contains the value **`+DD32`**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **`+DA2.0W`**. For 32-bit support on a PA-RISC platform, this flag contains the value **`+DA2.0N`**.

> **`+DD64`** Must be used to generate 64-bit code for HP-UX on IA64.
>
> **`+DD32`** Must be used to generate 32-bit code for HP-UX on IA64.
>
> **`+DA2.0W`**
> Must be used to generate 64-bit code for HP-UX on PA-RISC.
>
> **`+DA2.0N`**
> Must be used to generate 32-bit code for HP-UX on PA-RISC.

**`-Ae`** Enables HP ANSI extended mode.

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`

**`-c`** Perform compile only; no link. Compile and link are separate steps.

**Link options:**

**cc** Use the compiler as a front end for the linker.

**$EXTRA_CFLAG**
If the HP-UX platform is IA64 and 64-bit support is enabled, this flag contains the value **+DD64**; if 32-bit support is enabled, it contains the value **+DD32**. If the HP-UX platform is PA-RISC and 64-bit support is enabled, it contains the value **+DA2.0W**. For 32-bit support on a PA-RISC platform, this flag contains the value **+DA2.0N**.

- **+DD64** Must be used to generate 64-bit code for HP-UX on IA64.
- **+DD32** Must be used to generate 32-bit code for HP-UX on IA64.
- **+DA2.0W**
  Must be used to generate 64-bit code for HP-UX on PA-RISC.
- **+DA2.0N**
  Must be used to generate 32-bit code for HP-UX on PA-RISC.

**-o $1** Specify the executable program.

**$1.o** Specify the object file.

**utilci.o**
Include the utility object file for error checking.

**$EXTRA_LFLAG**
Specify the runtime path. If set, for 32-bit it contains the value `-Wl,+b$HOME/sqllib/lib32`, and for 64-bit: `-Wl,+b$HOME/sqllib/lib64`. If not set, it contains no value.

**-L$DB2PATH/$LIB**
Specify the location of the DB2 runtime shared libraries. For 32-bit: `$HOME/sqllib/lib32`; for 64-bit: `$HOME/sqllib/lib64`.

**-ldb2ci**
Link with the DB2CI library.

## DB2CI application compile and link options (Linux)

The compile and link options in this topic are recommended for building DB2CI applications with the GNU/Linux gcc compiler.

You can find the following options in the *DB2DIR*`/samples/db2ci/bldapp` batch file, where *DB2DIR* is the location where your DB2 copy is installed.

Compile and link options for bldapp

**Compile options:**

**gcc** The C compiler.

**$EXTRA_C_FLAGS**
Contains one of the following:

- `-m31` on Linux for zSeries® only, to build a 32-bit library;
- `-m32` on Linux for x86, x64 and POWER®, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

**-I$DB2PATH/include**
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`

**-c** Perform compile only; no link. Compile and link are separate steps.

**Link options:**

**`gcc`** Use the compiler as a front end for the linker.

**`$EXTRA_C_FLAGS`**
Contains one of the following:

- `-m31` on Linux for zSeries only, to build a 32-bit library;
- `-m32` on Linux for x86, x64 and POWER, to build a 32-bit library;
- `-m64` on Linux for zSeries, POWER, x64, to build a 64-bit library; or
- No value on Linux for IA64, to build a 64-bit library.

**`-o $1`** Specify the executable.

**`$1.o`** Include the program object file.

**`utilci.o`**
Include the utility object file for error checking.

**`$EXTRA_LFLAG`**
For 32-bit it contains the value "-Wl,-rpath,$DB2PATH/lib32", and for 64-bit it contains the value "-Wl,-rpath,$DB2PATH/lib64".

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`.

**`-ldb2ci`**
Link with the DB2CI library.

## DB2CI application compile and link options (Solaris)

The compile and link options in this topic are recommended for building DB2CI applications with the the Solaris C compiler.

You can find the following options in the *DB2DIR*`/samples/db2ci/bldapp` batch file, where *DB2DIR* is the location where your DB2 copy is installed.

Compile and link options for bldapp

**Compile options:**

**`cc`** Use the C compiler.

**`-xarch=$CFLAG_ARCH`**
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set as follows:

- "v8plusa" for 32-bit applications on Solaris SPARC
- "v9" for 64-bit applications on Solaris SPARC
- "sse2" for 32-bit applications on Solaris x64
- "amd64" for 64-bit applications on Solaris x64

**`-I$DB2PATH/include`**
Specify the location of the DB2 include files. For example: `$HOME/sqllib/include`

**`-c`** Perform compile only; no link. This script has separate compile and link steps.

**Link options:**

**`cc`** Use the compiler as a front end for the linker.

**`-xarch=$CFLAG_ARCH`**
This option ensures that the compiler will produce valid executables when linking with `libdb2.so`. The value for $CFLAG_ARCH is set to either "v8plusa" for 32-bit, or "v9" for 64-bit.

**`-mt`** Link in multi-thread support to prevent problems calling fopen.
**Note:** If POSIX threads are used, DB2 applications also have to link with `-lpthread`, whether or not they are threaded.

**`-o $1`** Specify the executable program.

**`$1.o`** Include the program object file.

**`utilci.o`**
Include the utility object file for error checking.

**`-L$DB2PATH/$LIB`**
Specify the location of the DB2 static and shared libraries at link-time. For example, for 32-bit: `$HOME/sqllib/lib32`, and for 64-bit: `$HOME/sqllib/lib64`

**`$EXTRA_LFLAG`**
Specify the location of the DB2 shared libraries at run-time. For 32-bit it contains the value "-R$DB2PATH/lib32", and for 64-bit it contains the value "-R$DB2PATH/lib64".

**`-ldb2ci`**
Link with the DB2CI library.

## DB2CI application compile and link options (Windows)

The compile and link options in this topic are recommended for building DB2CI applications with the Microsoft® Visual C++ compiler.

You can find the following options in the *DB2DIR*`\samples\db2ci\bldapp.bat` batch file, where *DB2DIR* is the location where your DB2 copy is installed.

Compile and link options for bldapp

**Compile options:**

**`%BLDCOMP%`**
Variable for the compiler. The default is `cl`, the Microsoft Visual C++ compiler. It can be also set to `icl`, the Intel® C++ Compiler for 32-bit and 64-bit applications, or `ecl`, the Intel C++ Compiler for Itanium® 64-bit applications.

**`-Zi`** Enable debugging information.

**`-Od`** Disable optimizations. It is easier to use a debugger with optimization off.

**`-c`** Perform compile only; no link.

**`-W2`** Set warning level.

**`-DWIN32`**
Compiler option necessary for Windows operating systems.

**Link options:**

**`link`** Use the linker.

**`-debug`** Include debugging information.

**`-out:%1.exe`**
Specify the executable.

**`%1.obj`** Include the object file.

**`db2ci.lib or db2ci64.lib`**
Link to the DB2CI library. For Windows 32-bit operating systems, use `db2ci.lib`. For Windows 64-bit operating systems, use `db2ci64.lib`.

Refer to your compiler documentation for additional compiler options.

# Appendix A. Overview of the DB2 technical information

DB2 technical information is available through the following tools and methods:

- DB2 Information Center
  - Topics (Task, concept and reference topics)
  - Help for DB2 tools
  - Sample programs
  - Tutorials
- DB2 books
  - PDF files (downloadable)
  - PDF files (from the DB2 PDF DVD)
  - printed books
- Command line help
  - Command help
  - Message help

**Note:** The DB2 Information Center topics are updated more frequently than either the PDF or the hardcopy books. To get the most current information, install the documentation updates as they become available, or refer to the DB2 Information Center at ibm.com.

You can access additional DB2 technical information such as technotes, white papers, and IBM Redbooks® publications online at ibm.com. Access the DB2 Information Management software library site at http://www.ibm.com/software/data/sw-library/.

### Documentation feedback

We value your feedback on the DB2 documentation. If you have suggestions for how to improve the DB2 documentation, send an e-mail to db2docs@ca.ibm.com. The DB2 documentation team reads all of your feedback, but cannot respond to you directly. Provide specific examples wherever possible so that we can better understand your concerns. If you are providing feedback on a specific topic or help file, include the topic title and URL.

Do not use this e-mail address to contact DB2 Customer Support. If you have a DB2 technical issue that the documentation does not resolve, contact your local IBM service center for assistance.

## DB2 technical library in hardcopy or PDF format

The following tables describe the DB2 library available from the IBM Publications Center at www.ibm.com/e-business/linkweb/publications/servlet/pbi.wss. English and translated DB2 Version 9.7 manuals in PDF format can be downloaded from www.ibm.com/support/docview.wss?rs=71&uid=swg2700947.

Although the tables identify books available in print, the books might not be available in your country or region.

© Copyright IBM Corp. 1993, 2010

The form number increases each time a manual is updated. Ensure that you are reading the most recent version of the manuals, as listed below.

**Note:** The *DB2 Information Center* is updated more frequently than either the PDF or the hard-copy books.

*Table 41. DB2 technical information*

| Name | Form Number | Available in print | Last updated |
|---|---|---|---|
| *Administrative API Reference* | SC27-2435-02 | Yes | September, 2010 |
| *Administrative Routines and Views* | SC27-2436-02 | No | September, 2010 |
| *Call Level Interface Guide and Reference, Volume 1* | SC27-2437-02 | Yes | September, 2010 |
| *Call Level Interface Guide and Reference, Volume 2* | SC27-2438-02 | Yes | September, 2010 |
| *Command Reference* | SC27-2439-02 | Yes | September, 2010 |
| *Data Movement Utilities Guide and Reference* | SC27-2440-00 | Yes | August, 2009 |
| *Data Recovery and High Availability Guide and Reference* | SC27-2441-02 | Yes | September, 2010 |
| *Database Administration Concepts and Configuration Reference* | SC27-2442-02 | Yes | September, 2010 |
| *Database Monitoring Guide and Reference* | SC27-2458-02 | Yes | September, 2010 |
| *Database Security Guide* | SC27-2443-01 | Yes | November, 2009 |
| *DB2 Text Search Guide* | SC27-2459-02 | Yes | September, 2010 |
| *Developing ADO.NET and OLE DB Applications* | SC27-2444-01 | Yes | November, 2009 |
| *Developing Embedded SQL Applications* | SC27-2445-01 | Yes | November, 2009 |
| *Developing Java Applications* | SC27-2446-02 | Yes | September, 2010 |
| *Developing Perl, PHP, Python, and Ruby on Rails Applications* | SC27-2447-01 | No | September, 2010 |
| *Developing User-defined Routines (SQL and External)* | SC27-2448-01 | Yes | November, 2009 |
| *Getting Started with Database Application Development* | GI11-9410-01 | Yes | November, 2009 |
| *Getting Started with DB2 Installation and Administration on Linux and Windows* | GI11-9411-00 | Yes | August, 2009 |

Table 41. DB2 technical information (continued)

| Name | Form Number | Available in print | Last updated |
|---|---|---|---|
| *Globalization Guide* | SC27-2449-00 | Yes | August, 2009 |
| *Installing DB2 Servers* | GC27-2455-02 | Yes | September, 2010 |
| *Installing IBM Data Server Clients* | GC27-2454-01 | No | September, 2010 |
| *Message Reference Volume 1* | SC27-2450-00 | No | August, 2009 |
| *Message Reference Volume 2* | SC27-2451-00 | No | August, 2009 |
| *Net Search Extender Administration and User's Guide* | SC27-2469-02 | No | September, 2010 |
| *Partitioning and Clustering Guide* | SC27-2453-01 | Yes | November, 2009 |
| *pureXML Guide* | SC27-2465-01 | Yes | November, 2009 |
| *Query Patroller Administration and User's Guide* | SC27-2467-00 | No | August, 2009 |
| *Spatial Extender and Geodetic Data Management Feature User's Guide and Reference* | SC27-2468-01 | No | September, 2010 |
| *SQL Procedural Languages: Application Enablement and Support* | SC27-2470-02 | Yes | September, 2010 |
| *SQL Reference, Volume 1* | SC27-2456-02 | Yes | September, 2010 |
| *SQL Reference, Volume 2* | SC27-2457-02 | Yes | September, 2010 |
| *Troubleshooting and Tuning Database Performance* | SC27-2461-02 | Yes | September, 2010 |
| *Upgrading to DB2 Version 9.7* | SC27-2452-02 | Yes | September, 2010 |
| *Visual Explain Tutorial* | SC27-2462-00 | No | August, 2009 |
| *What's New for DB2 Version 9.7* | SC27-2463-02 | Yes | September, 2010 |
| *Workload Manager Guide and Reference* | SC27-2464-02 | Yes | September, 2010 |
| *XQuery Reference* | SC27-2466-01 | No | November, 2009 |

Table 42. DB2 Connect-specific technical information

| Name | Form Number | Available in print | Last updated |
|---|---|---|---|
| *Installing and Configuring DB2 Connect Personal Edition* | SC27-2432-02 | Yes | September, 2010 |
| *Installing and Configuring DB2 Connect Servers* | SC27-2433-02 | Yes | September, 2010 |

Table 42. DB2 Connect-specific technical information (continued)

| Name | Form Number | Available in print | Last updated |
|---|---|---|---|
| *DB2 Connect User's Guide* | SC27-2434-02 | Yes | September, 2010 |

Table 43. Information Integration technical information

| Name | Form Number | Available in print | Last updated |
|---|---|---|---|
| *Information Integration: Administration Guide for Federated Systems* | SC19-1020-02 | Yes | August, 2009 |
| *Information Integration: ASNCLP Program Reference for Replication and Event Publishing* | SC19-1018-04 | Yes | August, 2009 |
| *Information Integration: Configuration Guide for Federated Data Sources* | SC19-1034-02 | No | August, 2009 |
| *Information Integration: SQL Replication Guide and Reference* | SC19-1030-02 | Yes | August, 2009 |
| *Information Integration: Introduction to Replication and Event Publishing* | GC19-1028-02 | Yes | August, 2009 |

## Ordering printed DB2 books

If you require printed DB2 books, you can buy them online in many but not all countries or regions. You can always order printed DB2 books from your local IBM representative. Keep in mind that some softcopy books on the *DB2 PDF Documentation* DVD are unavailable in print. For example, neither volume of the *DB2 Message Reference* is available as a printed book.

Printed versions of many of the DB2 books available on the DB2 PDF Documentation DVD can be ordered for a fee from IBM. Depending on where you are placing your order from, you may be able to order books online, from the IBM Publications Center. If online ordering is not available in your country or region, you can always order printed DB2 books from your local IBM representative. Note that not all books on the DB2 PDF Documentation DVD are available in print.

**Note:** The most up-to-date and complete DB2 documentation is maintained in the DB2 Information Center at http://publib.boulder.ibm.com/infocenter/db2luw/v9r7.

To order printed DB2 books:

- To find out whether you can order printed DB2 books online in your country or region, check the IBM Publications Center at http://www.ibm.com/shop/publications/order. You must select a country, region, or language to access publication ordering information and then follow the ordering instructions for your location.
- To order printed DB2 books from your local IBM representative:

1. Locate the contact information for your local representative from one of the following Web sites:
   - The IBM directory of world wide contacts at www.ibm.com/planetwide
   - The IBM Publications Web site at http://www.ibm.com/shop/publications/order. You will need to select your country, region, or language to the access appropriate publications home page for your location. From this page, follow the "About this site" link.
2. When you call, specify that you want to order a DB2 publication.
3. Provide your representative with the titles and form numbers of the books that you want to order. For titles and form numbers, see “DB2 technical library in hardcopy or PDF format” on page 403.

## Displaying SQL state help from the command line processor

DB2 products return an SQLSTATE value for conditions that can be the result of an SQL statement. SQLSTATE help explains the meanings of SQL states and SQL state class codes.

To start SQL state help, open the command line processor and enter:

```
   ? sqlstate or ? class code
```

where *sqlstate* represents a valid five-digit SQL state and *class code* represents the first two digits of the SQL state.
For example, `? 08003` displays help for the 08003 SQL state, and `? 08` displays help for the 08 class code.

## Accessing different versions of the DB2 Information Center

For DB2 Version 9.8 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r8/.

For DB2 Version 9.7 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/.

For DB2 Version 9.5 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9r5.

For DB2 Version 9.1 topics, the *DB2 Information Center* URL is http://publib.boulder.ibm.com/infocenter/db2luw/v9/.

For DB2 Version 8 topics, go to the *DB2 Information Center* URL at: http://publib.boulder.ibm.com/infocenter/db2luw/v8/.

## Displaying topics in your preferred language in the DB2 Information Center

The DB2 Information Center attempts to display topics in the language specified in your browser preferences. If a topic has not been translated into your preferred language, the DB2 Information Center displays the topic in English.

- To display topics in your preferred language in the Internet Explorer browser:
  1. In Internet Explorer, click the **Tools —> Internet Options —> Languages...** button. The Language Preferences window opens.

2. Ensure your preferred language is specified as the first entry in the list of languages.
   - To add a new language to the list, click the **Add...** button.

     **Note:** Adding a language does not guarantee that the computer has the fonts required to display the topics in the preferred language.
   - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
3. Refresh the page to display the DB2 Information Center in your preferred language.

- To display topics in your preferred language in a Firefox or Mozilla browser:
  1. Select the button in the **Languages** section of the **Tools —> Options —> Advanced** dialog. The Languages panel is displayed in the Preferences window.
  2. Ensure your preferred language is specified as the first entry in the list of languages.
     - To add a new language to the list, click the **Add...** button to select a language from the Add Languages window.
     - To move a language to the top of the list, select the language and click the **Move Up** button until the language is first in the list of languages.
  3. Refresh the page to display the DB2 Information Center in your preferred language.

On some browser and operating system combinations, you must also change the regional settings of your operating system to the locale and language of your choice.

## Updating the DB2 Information Center installed on your computer or intranet server

A locally installed DB2 Information Center must be updated periodically.

A DB2 Version 9.7 Information Center must already be installed. For details, see the “Installing the DB2 Information Center using the DB2 Setup wizard” topic in *Installing DB2 Servers*. All prerequisites and restrictions that applied to installing the Information Center also apply to updating the Information Center.

An existing DB2 Information Center can be updated automatically or manually:

- Automatic updates - updates existing Information Center features and languages. An additional benefit of automatic updates is that the Information Center is unavailable for a minimal period of time during the update. In addition, automatic updates can be set to run as part of other batch jobs that run periodically.
- Manual updates - should be used when you want to add features or languages during the update process. For example, a local Information Center was originally installed with both English and French languages, and now you want to also install the German language; a manual update will install German, as well as, update the existing Information Center features and languages. However, a manual update requires you to manually stop, update, and restart the Information Center. The Information Center is unavailable during the entire update process.

This topic details the process for automatic updates. For manual update instructions, see the "Manually updating the DB2 Information Center installed on your computer or intranet server" topic.

To automatically update the DB2 Information Center installed on your computer or intranet server:

1. On Linux operating systems,
   a. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `/opt/ibm/db2ic/V9.7` directory.
   b. Navigate from the installation directory to the `doc/bin` directory.
   c. Run the `ic-update` script:
      ```
      ic-update
      ```
2. On Windows operating systems,
   a. Open a command window.
   b. Navigate to the path where the Information Center is installed. By default, the DB2 Information Center is installed in the `<Program Files>\IBM\DB2 Information Center\Version 9.7` directory, where `<Program Files>` represents the location of the Program Files directory.
   c. Navigate from the installation directory to the `doc\bin` directory.
   d. Run the `ic-update.bat` file:
      ```
      ic-update.bat
      ```

The DB2 Information Center restarts automatically. If updates were available, the Information Center displays the new and updated topics. If Information Center updates were not available, a message is added to the log. The log file is located in `doc\eclipse\configuration` directory. The log file name is a randomly generated number. For example, `1239053440785.log`.

## Manually updating the DB2 Information Center installed on your computer or intranet server

If you have installed the DB2 Information Center locally, you can obtain and install documentation updates from IBM.

Updating your locally-installed *DB2 Information Center* manually requires that you:

1. Stop the *DB2 Information Center* on your computer, and restart the Information Center in stand-alone mode. Running the Information Center in stand-alone mode prevents other users on your network from accessing the Information Center, and allows you to apply updates. The Workstation version of the DB2 Information Center always runs in stand-alone mode. .
2. Use the Update feature to see what updates are available. If there are updates that you must install, you can use the Update feature to obtain and install them

   **Note:** If your environment requires installing the *DB2 Information Center* updates on a machine that is not connected to the internet, mirror the update site to a local file system using a machine that is connected to the internet and has the *DB2 Information Center* installed. If many users on your network will be installing the documentation updates, you can reduce the time required for individuals to perform the updates by also mirroring the update site locally and creating a proxy for the update site.

If update packages are available, use the Update feature to get the packages. However, the Update feature is only available in stand-alone mode.

3. Stop the stand-alone Information Center, and restart the *DB2 Information Center* on your computer.

**Note:** On Windows 2008, Windows Vista (and higher), the commands listed later in this section must be run as an administrator. To open a command prompt or graphical tool with full administrator privileges, right-click the shortcut and then select **Run as administrator**.

To update the *DB2 Information Center* installed on your computer or intranet server:

1. Stop the *DB2 Information Center*.
   - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click **DB2 Information Center** service and select **Stop**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv97 stop
     ```
2. Start the Information Center in stand-alone mode.
   - On Windows:
     a. Open a command window.
     b. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the `*Program_Files*\IBM\DB2 Information Center\Version 9.7` directory, where *Program_Files* represents the location of the `Program Files` directory.
     c. Navigate from the installation directory to the `doc\bin` directory.
     d. Run the `help_start.bat` file:

        ```
        help_start.bat
        ```
   - On Linux:
     a. Navigate to the path where the Information Center is installed. By default, the *DB2 Information Center* is installed in the `/opt/ibm/db2ic/V9.7` directory.
     b. Navigate from the installation directory to the `doc/bin` directory.
     c. Run the `help_start` script:

        ```
        help_start
        ```

   The systems default Web browser opens to display the stand-alone Information Center.
3. Click the **Update** button ( ). (JavaScript™ must be enabled in your browser.) On the right panel of the Information Center, click **Find Updates**. A list of updates for existing documentation displays.
4. To initiate the installation process, check the selections you want to install, then click **Install Updates**.
5. After the installation process has completed, click **Finish**.
6. Stop the stand-alone Information Center:
   - On Windows, navigate to the installation directory's `doc\bin` directory, and run the `help_end.bat` file:

     ```
     help_end.bat
     ```

     **Note:** The `help_end` batch file contains the commands required to safely stop the processes that were started with the `help_start` batch file. Do not use `Ctrl-C` or any other method to stop `help_start.bat`.

- On Linux, navigate to the installation directory's `doc/bin` directory, and run the `help_end` script:

  ```
  help_end
  ```

  **Note:** The `help_end` script contains the commands required to safely stop the processes that were started with the `help_start` script. Do not use any other method to stop the `help_start` script.

7. Restart the *DB2 Information Center*.
   - On Windows, click **Start** → **Control Panel** → **Administrative Tools** → **Services**. Then right-click **DB2 Information Center** service and select **Start**.
   - On Linux, enter the following command:

     ```
     /etc/init.d/db2icdv97 start
     ```

The updated *DB2 Information Center* displays the new and updated topics.

# DB2 tutorials

The DB2 tutorials help you learn about various aspects of DB2 products. Lessons provide step-by-step instructions.

## Before you begin

You can view the XHTML version of the tutorial from the Information Center at http://publib.boulder.ibm.com/infocenter/db2help/.

Some lessons use sample data or code. See the tutorial for a description of any prerequisites for its specific tasks.

## DB2 tutorials

To view the tutorial, click the title.

**"pureXML®" in** *pureXML Guide*
Set up a DB2 database to store XML data and to perform basic operations with the native XML data store.

**"Visual Explain" in** *Visual Explain Tutorial*
Analyze, optimize, and tune SQL statements for better performance using Visual Explain.

# DB2 troubleshooting information

A wide variety of troubleshooting and problem determination information is available to assist you in using DB2 database products.

**DB2 documentation**
Troubleshooting information can be found in the *Troubleshooting and Tuning Database Performance* or the Database fundamentals section of the *DB2 Information Center*. There you will find information about how to isolate and identify problems using DB2 diagnostic tools and utilities, solutions to some of the most common problems, and other advice on how to solve problems you might encounter with your DB2 database products.

**DB2 Technical Support Web site**
Refer to the DB2 Technical Support Web site if you are experiencing problems and want help finding possible causes and solutions. The Technical Support site has links to the latest DB2 publications, TechNotes,

Authorized Program Analysis Reports (APARs or bug fixes), fix packs, and other resources. You can search through this knowledge base to find possible solutions to your problems.

Access the DB2 Technical Support Web site at http://www.ibm.com/software/data/db2/support/db2_9/

## Terms and Conditions

Permissions for the use of these publications is granted subject to the following terms and conditions.

**Personal use:** You may reproduce these Publications for your personal, non commercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these Publications, or any portion thereof, without the express consent of IBM.

**Commercial use:** You may reproduce, distribute and display these Publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these Publications, or reproduce, distribute or display these Publications or any portion thereof outside your enterprise, without the express consent of IBM.

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the Publications or any information, data, software or other intellectual property contained therein.

IBM reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the Publications is detrimental to its interest or, as determined by IBM, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

IBM MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

# Appendix B. Notices

This information was developed for products and services offered in the U.S.A. Information about non-IBM products is based on information available at the time of first publication of this document and is subject to change.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

**The following paragraph does not apply to the United Kingdom or any other country/region where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web

© Copyright IBM Corp. 1993, 2010

sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information that has been exchanged, should contact:

IBM Canada Limited
U59/3600
3600 Steeles Avenue East
Markham, Ontario L3R 9Z7
CANADA

Such information may be available, subject to appropriate terms and conditions, including, in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems, and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements, or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility, or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious, and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Each copy or any portion of these sample programs or any derivative work must include a copyright notice as follows:

© (*your company name*) (*year*). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _*enter the year or years*_. All rights reserved.

## Trademarks

IBM, the IBM logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

The following terms are trademarks or registered trademarks of other companies

- Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- UNIX is a registered trademark of The Open Group in the United States and other countries.
- Intel, Intel logo, Intel Inside®, Intel Inside logo, Intel® Centrino®, Intel Centrino logo, Celeron®, Intel® Xeon®, Intel SpeedStep®, Itanium, and Pentium® are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.
- Microsoft, Windows, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

# Index

© Copyright IBM Corp. 1993, 2010

Printed in USA

SC27-2470-02

Spine information:

IBM DB2 9.7 for Linux, UNIX, and Windows **Version 9 Release 7** **SQL Procedural Languages: Application Enablement and Support**