Db2 11.1 for Linux, UNIX, and Windows

SQL PL

Db2 11.1 for Linux, UNIX, and Windows

SQL PL

Notice regarding this document

This document in PDF form is provided as a courtesy to customers who have requested documentation in this format. It is provided As-Is without warranty or maintenance commitment.

Contents

Figures

Tables

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.

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

You can use SQL PL to implement procedural logic 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 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 ATOMIC
  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.

Error location in the SQLCA structure

When an error occurs during the compilation or execution of an SQL PL object, the SQLCA structure that is returned by the Db2 database manager contains information that helps in identifying the origin of the error.

Compile-time errors

When an error is raised during the compilation of an SQL PL object (either inlined or compiled), the sqlerrd(3) field in the returned SQLCA structure contains the line number of the error as reported by the Db2 SQL compiler.

In the following example, the procedure myproc contains an invalid statement in line 5. The statement is invalid because the data types of the source and target portions of the SET statement do not match.

```
create procedure myproc (parm1 integer)
 begin
   declare var1 date;
   set var1 = parm1;
 end @
```

Assuming that this procedure is stored in file `script1.db2`, an attempt is made to process this file using the command line processor (CLP) with the following command:

```
db2 -td@ -a -f script1.db2
```

Where using the *-a* option causes the CLP to display the SQLCA. The result would be similar to the following information:

```
SQLCA Information

 sqlcaid : SQLCA      sqlcabc: 136   sqlcode: -408
 sqlerrml: 4
 sqlerrmc: VAR1
 sqlerrd : (1) 0         (2) 0               (3) 5
           (4) 0         (5) 0               (6) 0
 ...
 sqlstate: 42821
```

The value of 5 in the sqlerrd(3) field indicates that the error was in line 5 of the CREATE PROCEDURE statement.

The CLP was used as the client interface for this example because it provides a simple way of showing the SQLCA structure that results from the execution of a statement. However, when compiling SQL PL statements using the CLP, it is not necessary to display the SQLCA structure to identify the line number of an error. The CLP automatically displays the line number when it formats the error message. For the created procedure (myproc), an invocation of the CLP that did not include the *-a* option would return the following message, which shows that the error was in line 5:

```
SQL0408N  A value is not compatible with the data type of its assignment
target. Target name is "VAR1".  LINE NUMBER=5.  SQLSTATE=42821
```

Runtime errors

When an error is raised during the execution of a compiled SQL PL routine or trigger, the SQLCA structure that is returned by the Db2 database manager contains both the line number of the statement that caused the error and a numeric value that uniquely identifies the SQL PL routine or trigger that contains that statement.

In the following example, a statement creates the table table1 with one integer column. A second statement creates the procedure appdev.proc2 with two integer parameters, *parm1* and *parm2*, that inserts into the table the value that results from dividing *parm1* by *parm2*. The CALL statement with arguments 1 and 0 results in a division-by-zero runtime error.

```
create table table1 (col1 integer) @

create procedure appdev.proc2(in parm1 integer, parm2 integer)
specific appdev_proc2
begin
   insert into table1  (parm1 / parm2);
end @

call proc2(1, 0) @
```

Assuming that these statements are stored in file `script2.db2.`, an attempt is made to process this file by using the CLP with the following command:

```
db2 -td@ -a -f script2.db2
```

After executing the call, the result from CLP would be similar to the following output:

```
SQLCA Information

 sqlcaid : SQLCAM    sqlcabc: 136   sqlcode: -801
```

```
...
sqlerrd : (1) 0         (2) 0               (3) 4
          (4) 13152254  (5) 0               (6) 0

...
sqlstate: 22012
```

The value of sqlcode -801 in the result corresponds to division by zero (SQLSTATE 22012). Similar to compile-time errors, the value in the sqlerrd(3) field indicates the line number within the SQL PL object where the error originated (line number 4 in this case). Additionally, the value in the sqlerrd(4) field contains an integer value that uniquely identifies that SQL PL object. The SYSPROC.GET_ROUTINE_NAME procedure can be used to map the unique identifier to the name of the object that raised the error. This procedure takes as an input parameter the value in the sqlerrd(4) field and returns information in five output parameters, as shown in the following example:

```
db2 CALL SYSPROC.GET_ROUTINE_NAME(13152254, ?, ?, ?, ?, ?)

  Value of output parameters
  --------------------------
  Parameter Name  : TYPE
  Parameter Value : P

  Parameter Name  : SCHEMA
  Parameter Value : APPDEV

  Parameter Name  : MODULE
  Parameter Value : -

  Parameter Name  : NAME
  Parameter Value : PROC2

  Parameter Name  : SPECIFIC_NAME
  Parameter Value : APPDEV_PROC2

  Return Status = 0
```

The value 'P' for the TYPE parameter indicates that the object is a procedure. Because the object does not belong to a module, the MODULE parameter is NULL.

Note: The SQLCA structure does not contain line number information for errors that are raised during the execution of inline SQL PL objects.

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:

- 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 a statistical view that is associated with an expression-based index, or columns of a statistical view that is associated with an expression-based index.

- 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.

Before you begin

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

Procedure

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.

Results

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

Example

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;
```

What to do next

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 following set of examples 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
  - Nested types in local types declared inside an SQL routine
- 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 10
- Read: "Restrictions on the row data type" on page 11

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 10
- Read: “Restrictions on the row data type” on page 11

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 previously inserts into each column in the table, the corresponding row field value. Thus the previously shown INSERT statement 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.

Before you begin

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.

About this task

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.

Procedure

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.

Example

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

```
DROP TYPE simpleRow;
```

What to do next

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: `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: `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:
`BEGIN`
`DECLARE mySimpleA simple;`
 	
`SET mySimpleA[100] = 123;`
 	
`END`
 	
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:
	`BEGIN`
	`DECLARE myAssocA assoc;`
	
	`SET myAssocA[100] = 123;`
	`END`
	
	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.

- The use of arrays with row type elements in autonomous routines is not supported, if any of the row fields belonging to the element type are arrays or other row types.

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.

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 begin

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

About this task

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 27

Procedure

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.

Results

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

Example

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.

What to do next

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 begin

Before you create a local variable of type row:

- Read: Array data types
- Read: "Restrictions on the array data type" on page 27
- Read: "Creating an array data type (CREATE TYPE statement)" on page 28
- Ensure that you have the privileges required to execute the DECLARE statement.

About this task

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

Procedure

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.

Results

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.

What to do next

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.

Assigning array values using the subindex and literal values:

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

Before you begin

- Read: “Ordinary array data type” on page 26
- Read: “Restrictions on the array data type” on page 27
- Privileges required to execute the SET statement

About this task

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.

Procedure

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.

Example

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;
```

What to do next

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.

Retrieving array values using an index:

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

Before you begin

The following are prerequisites to this task:

- Read: “Ordinary array data type” on page 26
- Read: “Restrictions on the array data type” on page 27
- Privileges required to execute the SET statement or any SQL statement that contains the array reference

About this task

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.

Procedure

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.

Example

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

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

What to do next

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.

Before you begin

- Read: “Ordinary array data type” on page 26
- Read: “Restrictions on the array data type” on page 27
- Privileges required to execute the SET statement

About this task

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.

Procedure

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.

Results

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

Example

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

```
SET card = CARDINALITY(arrayName);

SET maxcard = MAX_CARDINALITY(arrayName);
```

What to do next

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 (ARRAY_FIRST, ARRAY_LAST functions):

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

Before you begin

- Read: “Ordinary array data type” on page 26
- Read: “Restrictions on the array data type” on page 27
- Privileges required to execute the SET statement

About this task

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

Procedure

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 ARRAY_FIRST or ARRAY_LAST function and within the required brackets, the name of the array.

Results

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.

Example

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 = ARRAY_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[ARRAY_FIRST(A)]
```

What to do next

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 ARRAY_PRIOR and ARRAY_NEXT functions.

Before you begin

- Read: "Ordinary array data type" on page 26
- Read: "Restrictions on the array data type" on page 27
- Privileges required to execute the SET statement

About this task

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

Procedure

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_NEXT or ARRAY_PRIOR function and within the required brackets, the name of the array.
2. Execute the SET statement.

Example

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 = ARRAY_FIRST(phones);
```

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

```
SET nextPhone = ARRAY_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[ARRAY_NEXT(phones, nextPhone)];
```

What to do next

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.

Before you begin

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

About this task

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

Procedure

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.

Results

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

Example

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'
```

What to do next

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.

Before you begin

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

About this task

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

Procedure

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.

Results

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

Example

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'
```

What to do next

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.

Before you begin

- Read: "Ordinary array data type" on page 26
- Read: "Restrictions on the array data type" on page 27
- Privileges required to execute the IF statement or any SQL statement in which the ARRAY_EXISTS function is referenced.

About this task

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

Procedure

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.

Example

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;
```

What to do next

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 following examples 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 previously listed 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.
- The use of arrays with row type elements in autonomous routines is not supported, if any of the row fields belonging to the element type are arrays or other row types.

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.

Before you begin

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

About this task

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

Procedure

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

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)];
```

What to do next

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 types to 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.

Before you begin

- 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

About this task

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

Procedure

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

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.

What to do next

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.

Before you begin

- 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.

About this task

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.

Procedure

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

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.

What to do next

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 built-in 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 built-in 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.

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.

Before you begin

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 built-in 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))@
```

About this task

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.

Procedure

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.

Results

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

Example

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@
```

What to do next

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.

Before you begin

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;
```

About this task

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.

Procedure

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 built-in 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.

Results

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

What to do next

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.

About this task

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.

About this task

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.

Before you begin

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.

About this task

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.

Procedure

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.

Results

The statement should compile successfully.

Example

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

```
connect to sample %

set serveroutput on %

create or replace procedure rowcount_test()
language sql
begin
  declare rows_fetched bigint;
  declare Designers cursor;
  declare first anchor to employee.firstnme;
  declare last anchor to employee.lastname;

  set Designers = cursor for select firstnme, lastname
    from employee where job = 'DESIGNER' order by empno asc;

  open Designers;

  fetch Designers into first, last;
  call dbms_output.put_line(last || ', ' || first);
  fetch Designers into first, last;
  call dbms_output.put_line(last || ', ' || first);

  set rows_fetched = CURSOR_ROWCOUNT(Designers);
  call dbms_output.put_line(rows_fetched || ' rows fetched.');
```

```
  close Designers;
end %

call rowcount_test() %

connect reset %
terminate %
```

What to do next

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 SYSCURSOR)
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 SYSCURSOR;

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

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

END @

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

When this 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 CURSOR RETURNS myRowType
DB20000I  The SQL command completed successfully.

CREATE PROCEDURE P(IN pempNo VARCHAR(8),OUT pcv1 SYSCURSOR)
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 SYSCURSOR;

  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
```

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.

SQL routines 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 the “Overview of routines” topic. With that knowledge you can then learn more about SQL routines 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

CREATE statements can be used to create SQL routines and store the routine-body login within the statement.

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.

About this task

As long as SQL routines can meet your requirements, you should choose SQL routines over external routines. Use external routines when you must implement complex logic, or access files or scripts on a database server.

Procedure

- 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.

Determining when to use SQL procedures or SQL functions

SQL Procedural Language (SQL PL) is a language extension of SQL that consists of statements and language elements that you can use to implement procedural logic in SQL statements. You can implement logic with SQL PL by using SQL procedures or SQL functions.

Procedure

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.

Results

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

You can rewrite simple SQL procedures as SQL user-defined functions to maximize performance in the database management system.

About this task

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.

Procedure

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.

Example

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 “Overview of SQL procedures” topic.

Features of SQL procedures

Implementing SQL procedures can play an essential role in database architecture, database application design, and in database system performance.

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 the features listed previously.

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.

Parts of SQL procedures:

There are many parts that make up an SQL procedure. Each part is essential to ensure that the procedure is executed properly.

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 previously:

```
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;
     END A;

   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:

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.

SQL procedures support parameters for the passing of SQL values into and out of procedures.

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. 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, and others).
- 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

You can specify untyped parameter markers in selected locations of an SQL statement. Table 1 lists the restrictions on untyped 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

You can use a standard interface such as CLI/ODBC, JDBC, and ADO.NET to access the database. 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	Db2data 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:

You can use the SQLCODE and SQLSTATE variables to display error messages and codes that might occur during SQL procedure execution.

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. You can 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:

You can use compound statement to introduce a new scope in which variables might or might not be available for use.

SQL procedures can contain one or more compound statements. They can be introduced in serial or can be nested within another compound statement.

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
```

SQL PL logic elements in the SQL-procedure body:

SQL Procedural Language (SQL PL) provides support for variables and flow of control statements, which can be used to control the sequence of statement execution.

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.

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 previous two examples 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 this example 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 it 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 this example 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 (compiled) 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.

Before you begin

You must have the authority to create an SQL procedure.

Procedure

To return a result set from an SQL procedure:

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

Example

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.

Before you begin

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.

Procedure

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 previously) 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.

About this task

The CREATE PROCEDURE statement can be executed using graphical development environment tools or by directly executing the statement from the Db2 Command Line Processor (CLP), a Db2 Command Window, 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:

Using the command line to create SQL procedures can be faster than using graphical developmental environment tools.

Before you begin

- 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.

Procedure

- 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 (compiled) statements can be customized via a Db2 registry variable or some SQL procedure routines.

About this task

To customize precompile and bind options for compiled SQL objects, set the instance-wide Db2 registry variable, **DB2_SQLROUTINE_PREPOPTS**. For example:

```
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 following CLP scripts. 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 the Db2 instance 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 it is 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

When an SQL procedure is created, the SQL queries in the procedure body are separated 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 Db2optimizer for that query.

Overview of how SQL PL and inline SQL PL are compiled

Before discussing how to improve the performance of SQL procedures, we should discuss how they are compiled upon the execution of the CREATE PROCEDURE statement.

When an SQL procedure is created, the SQL queries in the procedure body are separated 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 database engine. SQL procedures run in *unfenced mode*, meaning they run in the same addressing space as the database 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 the database engine. 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 in the following example:

```
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 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 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. 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.

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 previously. 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 typically produce the best performance. However, they do not support some SQL PL language features and data types that compiled SQL functions support. These features include:

- the CASE statement
- the REPEAT statement
- Cursor processing
- Dynamic SQL
- Condition handlers

SQL functions that are declared in SQL PL modules are always compiled.

PL/SQL functions are always compiled. While inlined functions can be referenced in PL/SQL code, they cannot be declared using PL/SQL. They must be declared using SQL PL.

For all other cases, the syntax of the CREATE FUNCTION statement determines whether an SQL function is inlined or compiled.

- An SQL PL function with an atomic body is an inlined function. An SQL function is atomic if it contains a single RETURN statement, or consists of a compound statement that begins with the keywords BEGIN ATOMIC.
- An SQL PL function that is non-atomic is a compiled function. An SQL PL function is non-atomic if it contains a compound statement that begins with the keywords BEGIN or BEGIN NOT ATOMIC.

Examples

- Inlined functions

```
create function F1 (N integer)
returns integer
deterministic
no external action
contains sql
return N * 10
@

create function F2 (N integer)
returns integer
deterministic
no external action
```

```
contains sql
begin atomic
  if N < 5 then
    return N * 10
  else
    return N * 20
  end if;
end
@
```

- Compiled functions

```
create function F3 (N integer)
returns integer
deterministic
no external action
contains sql
begin
  if N < 5 then
    return N * 10
  else
    return N * 20
  end if;
end
@
```

```
create function F4 (N integer)
returns integer
deterministic
no external action
contains sql
begin not atomic
  if N < 5 then
    return N * 10
  else
    return N * 20
  end if;
end
@
```

Restrictions on SQL functions:

Be aware of 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, LONG VARCHAR data type, and LONG VARGRAPHIC data type. The XML data type is not supported in Version 10.1. The support for XML data type starts in Version 10.1 Fix Pack 1.
- In this version, use of the DECLARE TYPE statement within compiled SQL functions is not supported.
- Compiled SQL functions (including PL/SQL functions) must not contain references to federated objects.

- Compiled SQL functions (including PL/SQL functions) that modify SQL data can only be used as the only element on the right side of an assignment statement that is within a compound SQL (compiled) statement.
- If a table contains a generated column expression in which the user-defined function is a compiled compound SQL, then you cannot use the **LOAD** utility to insert data into the table.

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 begin

Before you create an SQL function:

- Read: "SQL functions" on page 97
- Read: "Features of SQL functions" on page 97
- Ensure that you have the privileges required to execute the CREATE FUNCTION (scalar) statement.

About this task

Restrictions

See: "Restrictions on SQL functions" on page 99

Procedure

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.

Results

The CREATE FUNCTION (scalar) statement should execute successfully and the scalar function should be created.

Example

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@
```

What to do next

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 begin

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 99

Procedure

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.

Results

The CREATE FUNCTION (table) statement should execute successfully and the table function should be created.

Example

Example 1

The following is an example of an 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 an SQL table function:

```
CREATE TABLE t1(pk INT, c1 INT, date DATE)

CREATE TABLE t1_archive LIKE T1%

CREATE FUNCTION archive_tbl_t1(ppk INT)
  RETURNS TABLE(pk INT, c1 INT, date DATE)
LANGUAGE SQL
MODIFIES SQL DATA
BEGIN ATOMIC

  DECLARE c1 INT;

  DECLARE date DATE;

  SET (c1, date) = (SELECT c1, date FROM OLD TABLE(DELETE FROM t1 WHERE t1.pk = ppk));

  INSERT INTO T1_ARCHIVE VALUES (ppk, c1, date);

  RETURN VALUES (ppk, c1, date);
END%
```

What to do next

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 in the following paragraphs.

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.

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.

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 begin

Before you create a compound statement:

- Read: "Compound statements" on page 103
- Ensure that you have the privileges required to execute the Compound statement.

Procedure

1. Define a compound SQL statement.
2. Execute the compound SQL statement from a supported interface.

Results

If executed dynamically, the SQL statement should execute successfully.

Example

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.

Index

R

S

U

V

W

Printed in USA