New SQL PL data types - with use cases you probably did not think of!

Peter Vanroose
ABIS Training & Consulting

Session Code: F4

Monday, 14 November 2016, 16:30 - 17:30 | Platform: Cross Platform

Abstract:

SQL PL has been available for some time; features are added with each new release of DB2. Understand how new data types like anchored types and arrays simplify your code and make it more efficient. Discover how SQL arrays can be used even outside SQL PL - yes, even in standard SQL statements! Learn when you should consider using them. And we will tell you - yes, we are also Oracle experts - what IBM should provide us with in a next release.

Objective 1: Introduce more complex SQL PL data types.

Objective 2: Explain how complex SQL PL data types can be used to facilitate writing SQL PL code.

Objective 3: Explain how complex SQL PL data types can be used to increase code efficiency - application performance!

Objective 4: Explain SQL arrays in detail - including use cases (standard SQL and SQL PL)!

Objective 5: Introduce some nice-to-have features - both from a flexibility and efficiency point of view.

Summary - Topics covered:

- Part I: SQL PL
  - what?
  - why?
  - where & how?
  - Application programming with DB2
    - possibilities
    - position of SQL PL
- Part II: New and/or less-known SQL PL possibilities
  - ARRAY datatype(s)
  - ROW datatype
  - Differences between DB2 for z/OS & DB2 for LUW ?
    - will focus on the similarities!
- Part III: Scenario's, Usage considerations, Nice to Haves

This presentation consists of three main parts, where the first part is kind of introductory, consisting of necessary background for the main topics of this presentation, in the other two parts.

Part I - SQL PL
=> Context sketch - bringing you up to speed

1. What is SQL PL ?
   - a programming language, but ... a pecular one!
2. Where can it be used ?
   - stored procedures & triggers
   - used-defined functions: scalar functions / table functions / row functions
   - stand-alone "compound SQL" (only LUW)
3. How does it look like ?
   - brief sketch of the structure
     - BEGIN...END blocks
     - the four subsections
4. How does it integrate with
   - SQL DML (SELECT, INSERT, UPDATE, DELETE) ?
   - applications using DB2 ? (Java, COBOL, C, REXX, PHP, ...)

“Table of contents” for part 1: what is SQL PL, where do you encounter it, what are its most important syntactic aspects?

The subsequent slides do strictly not follow this table of contents: the different aspects are introduced through some example program code.

SQL PL - a brief overview

- compound statement block:

```
BEGIN
  <local variable declarations>
  <local cursor declarations>
  <local handler declarations>
  <SQL PL statements>
END ;     /* note the ";" ! */
```

- variable declaration:

```
DECLARE <var-name>   <data-type>  [ DEFAULT <value> ] ;
```

- cursor declaration:

```
DECLARE <name>  CURSOR FOR  <SELECT-statement> ;
```

- handler declaration:

```
DECLARE  { CONTINUE | EXIT | UNDO }  HANDLER FOR
 {  SQLSTATE <sqlstate-value>
 |  SQLEXCEPTION  |  SQLWARNING  |  NOT FOUND   }
<single SQL PL statement> ;
```

SQL PL: the syntax.
It is very important to realize that SQL PL consists of BEGIN..END blocks which can be nested,
and that every block can contain up to four “subsections” that must appear in the correct order.
For the sake of this presentation, only the first and the last subsection are of importance.

SQL PL - a brief overview (cont'd)

- **ASSIGNMENT statement:**

```
SET <var-name> = <scalar-expression> ; /* incl. scalar SELECT */
```

- **IF statement:**

```
IF     <condition> THEN    <SQL PL statement(s)> ;
ELSEIF <condition> THEN    <SQL PL statement(s)> ;
ELSE                       <SQL PL statement(s)> ;
END IF ;
```

- **WHILE...DO statement:**

```
WHILE <condition> DO
  <SQL PL statement(s)> ;
END WHILE
```

- **FOR statement:**

```
FOR <name> AS <SELECT-statement> DO
  <SQL PL statement(s)>
END FOR
```

- **Also available: CASE stmt; REPEAT...UNTIL; ITERATE; LOOP; LEAVE**

The 4th subsection consists of run-time statements. These can in turn be BEGIN..END blocks (not mentioned here, but see the examples further on), SQL (DML/DDL) statements, or one of the “procedural” statements mentioned: assignment, if/then/else, while, and a few other ones (less important for the rest of this presentation).

SQL PL: a first example

Add a given product to the (possibly new) order with given order nr.

```
CREATE PROCEDURE add_prod_to_order(IN  ordno   INT,
                                   IN  prodno  INT,
                                   OUT status  VARCHAR(32))
BEGIN
  DECLARE detailno SMALLINT;
  DECLARE exists   INT;
  DECLARE sqlcode  INT;

  SET  exists  =  (SELECT count(*) FROM orders WHERE orid = ordno);
  IF exists = 0  OR  exists IS NULL  THEN
    INSERT INTO orders(orid) VALUES (ordno);
    IF sqlcode <> 0 THEN ... END IF;
  END IF;
  SELECT max(odno)+1 INTO detailno FROM orderdetails WHERE od_orid = ordno;
  INSERT INTO orderdetails (od_orid, odno, od_prid)
    VALUES (ordno, detailno, prodno);
  IF sqlcode <> 0 THEN ... END IF; /* whatever exception handling wanted */
END
```

An example of some SQL PL syntax, and at the same time a partial answer to the question “where can it be used”, viz. as the body of a stored procedure.

The example shows how subsections 1 and 4 are written (one before the other, but no “visual” boundary), how local variables (and actually also parameters) are declared, and how the most important kinds of statements are to be used:

- assignment
- SQL SELECT (here actually a scalar subselect used as RHS for the assignment)
- SQL SELECT INTO
- SQL INSERT
- IF..END IF
- the special variable SQLCODE (was not mentioned before -- needs some explanation: assigned to by the database engine responsible for the SQL DML & DDL statements)

Note: the implementation is not complete w.r.t. interfacing with the client: either a return status (success, failure, error message, ...) could be returned through an OUT parameter, or (preferably) exception handing (throwing an exception when failure or warning is to be passed back to the caller) could be used. Going into exception handling in SQL PL is out of the scope of this presentation.

SQL PL: a second example

For a given order nr., list the names of the products ordered.

```
CREATE FUNCTION products_of_order (ordno INT)
RETURNS varchar(32700)
BEGIN
  DECLARE productlist  VARCHAR(32700)  DEFAULT '';
  FOR c AS
    SELECT prname FROM products
     WHERE prid IN (SELECT od_prid FROM orderdetails WHERE od_orid = ordno)
     ORDER BY prname
  DO
    SET productlist = productlist || ', ' || prname;
  END FOR;
  SET productlist = substr(productlist, 3);
  IF productlist = '' THEN
    SET productlist = '*** No products ordered! ***';
  END IF;
  RETURN productlist;
END
```

A second example, with a second answer to the question “where can it be used”: in a user-defined (scalar) function.

And with an example of a very interesting kind of statement: FOR

Also important in the light of the ROW datatype to be discussed later, since there are some similarities...

Notice (also in the previous example) how builtin scalar functions (like substr) and predicates (like IS NULL) can be used in SQL PL, not just in SQL DML statements like a SELECT.

SQL PL: a third example

For a given order nr., return the list of ordered products & their quantities.

```
CREATE FUNCTION prods_of_order (ordno INT)
RETURNS TABLE (prodname VARCHAR(32), quantity INT)
  RETURN
    SELECT prname, prquantity FROM products
     WHERE prid IN (SELECT od_prid FROM orderdetails WHERE od_orid = ordno)
```

Stored procedures, functions or triggers implemented in SQL PL
don’t differ from those implemented in an other programming language
from the point of view of the “user”:
- CALLing a stored procedure: pass the input values, and pass parameter markers for output values
- using a user-defined function: pass the input values, plug in the result (scalar or table) where wanted
- “triggering” a trigger: is 100% transparent to the user; side effect of an INSERT/UPDATE/DELETE stmt

In the next part, it will become clear that the “new” structured datatypes introduce a situation where (some) SQL PL implemented routines *are* different, though: some stored procedures / functions will no longer be usable from some client environments; and by their “signature” they give away that their implementation must be SQL PL !

Using an SQL PL program

To "*launch*" a **stored proc** (written in SQL PL):

- from COBOL / Java / REXX / C / PHP ... (DB2 client) program:

```
EXEC SQL  CALL add_prod_to_order( :hv1, 500, :hv2 )
```

(param. markers (host variables) or constants)

- from within SQL PL (function/proc/trigger): **CALL** statement
- from LUW CLI:

```
db2 "CALL add_prod_to_order( 27, 500, ? )"
```

To use a **user-defined function** (written in SQL PL):

- in an SQL statement => *any* SQL client ! (incl. e.g. SPUFI)

```
SELECT products_of_order(27) FROM sysibm.sysdummy1;
SELECT * FROM TABLE (prods_of_order(27)) AS p  ORDER BY 1;
```

A stored procedure can only be CALLed
- from an SQL client connected to DB2
- but not e.g. SPUFI
- using the CALL statement (SQL) passing the parameter list (correct signature) between parentheses
   => IN parameters are passed from client to DB2; these can either be client-side constants or expressions, or host variables (parameter markers) in which case their content (at the time of the CALL) is sent.
   => OUT parameters are passed from DB2 to client; they must be specified with parameter markers (host variables): a “?” in a dynamic SQL context (e.g. JDBC or PHP) or the name of a variable preceded by a “:” in a static SQL context (e.g. COBOL, SQLJ, or C).
   => INOUT parameters are first passed from client to DB2, then filled back in by DB2; they must be specified with parameter markers.

A user-defined scalar function can be used in any context where a built-in scalar function is used:
- in the SELECT clause list of a SELECT statement
- as part of an expression in e.g. the SELECT or WHERE clauses, or in an UPDATE or INSERT rhs
- in SQL PL: as rhs of the SET statement, in an expression, or in a predicate for IF or WHILE

A user-defined table function can be used in the FROM clause of the SELECT statement
- don’t forget to use the TABLE(...) construct, to avoid a syntax error at the “(”
- don’t forget to specify a table alias name (“p” in the example)
- the column names and their datatypes are as specified in the CREATE statement.

Part II - composite data types: Arrays & Rows
=> only available within SQL PL

1. the "ordinary" basic *array* datatype
   - what is it
   - how to use it
2. the *associative array* datatype
   - what is it
   - how to use it
3. interesting use cases:
   - result table into array
   - array into table expression
   - replacement for cursors ...
4. (LUW only) *anchored* datatypes and the *ROW* datatype

“Table of contents” for part 2.

Now we’ll follow this chonology more closely.

Here and there, some “flash forwards” can be expected, especially to give examples of use, and to compare the complex datatypes mentioned with similar constructs in other environments than DB2, notably in programming languages like COBOL or Java, and in other database systems like Oracle.

Essentially, three datatypes (or “data structures”) are treated:

- ordinary arrays
- associative arrays
- rows

1. The basic “array” datatype

- New data type in DB2 11 for z/OS
  - DB2 LUW: since version 9.5
- ARRAY = “*vertical* list of objects of the same type”
  (most programming languages have arrays)
- is defined in terms of a built-in datatype, e.g. varchar(24):

```
CREATE TYPE array_txt  AS  varchar(24)   ARRAY  [ 1000 ] ;

SELECT name, sourcetype, length, metatype, array_length
FROM   sysibm.sysdatatypes  WHERE  definer = USER ;
```

```
NAME        SOURCETYPE   LENGTH       METATYPE   ARRAY_LENGTH(*)
----------  -----------  -----------  ---------  ------------
ARRAY_TXT   VARCHAR               24  A                  1000
                                                (*) no underscore on z/OS
```

- leaving out max. size specification (1000) => “unlimited” (maxint)

```
CREATE TYPE array_txt1  AS  varchar(24)  ARRAY  [ ] ;
```

Definition of an array variable:
- refer to p.232 of the SQL Reference for DB2 11 for z/OS (SC19-4066-06)
- and p.108 of the SQL Reference Volume 1 for DB2 10.5 for LUW (SC27-5509-00)

Currently, there are (unfortunately) no special registers nor global session variables of data type "array" in DB2 for z/OS, only in DB2 for LUW.
=> could be an interesting way of caching an intermediate result (table) within a session,
without the need to transfer it to/from the client, even across multiple calls of stored procedures.

Is similar to the "ARRAY" concept in other programming languages:
- Java, PHP, Perl, ... (including the [...] notation and its dynamic nature)
- Oracle PL/SQL (but very different in terms of dynamic allocation!) -- also: different notation () not []
- C : non-dynamic!
- COBOL: there it's called a "table"; again non-dynamic
on the other hand, COBOL tables typically have more than 1 column... (cf. the ROW type later)

The attribute "vertical" is explicitly mentioned to distinguish it from a subsequent "horizontal" list where the elements will no longer need to be of the same datatype.
"Vertical" refers to single columns of a table: also there, the cells must be of the same datatype.

Note that the datatype for the cells must be a built-in (scalar) datatype, *excluding XML* (but including e.g. CLOB)
In DB2 for LUW the cell datatype could additionally be a "structure" (see further on)

The basic “array” datatype: how to use it

- Only a *variable* in **SQL PL** can be of array type:

```
BEGIN   DECLARE my_list array_txt;     END @
```

- (Also registers or global (session) variables *could* be arrays ...)
- IN & OUT *parameters* & *return values* of procedures/functions
  (**this is an important use case!**)   (z/OS: *not* for table function params)

```
==> effectively limits its use to SQL PL !
```

- Filling an array: element-wise, or “as a whole” (constructor):

```
SET my_list[23] = 'value';  -- index from 1 up to 1000
SET my_list[ i ] = my_list [ i-1 ] ;  -- expression as index
SET my_list = ARRAY['value1', 'value2', 'value3', ...];
SET my_list = ARRAY[ SELECT expr FROM ... ];
SET my_list = my_other_list;
```

Assignment to arrays:

- assignment to an *element* a[i] of an array: identical to assignment to an “ordinary” variable:
    * with the SET statement
    * with SELECT ... INTO a[i] ...
    * by passing a[i] as OUT or INOUT parameter in a procedure call
- assignment to an array *as a whole*: by using the ARRAY [...] “constructor”
  three variants:
    - copy the content of one array as a whole to an other array
    - with a comma-separated list of (explicit) values
    - with a “fullselect” returning a single column

  in the latter case, indexes 1, 2, 3, etc. will be filled with the rows of the fullselect result table,
  in the order as if the fullselect were passed to a cursor and opened/fetched
  (possibly with an explicit ORDER BY)

- the three variants can be used in:
    - an explicit SET command
    - an implicit assignment, when passing an IN parameter from caller to callee,
      or an OUT parameter or the return value from callee to caller
    - the SELECT ... INTO statement with a multi-row select using the ARRAY_AGG function

2. Associative arrays

- "variant" of the basic array:
  - still a "list of objects of the same type"
  - still defined in terms of a built-in datatype, e.g. varchar(24)
  - BUT the index need not be integer! E.g.:

```
CREATE TYPE array_txt2 AS  varchar(24) ARRAY [ varchar(9) ] ;

SELECT name, sourcetype, length, metatype, array_length
FROM   sysibm.sysdatatypes  WHERE  definer = USER ;
```

```
NAME        SOURCETYPE   LENGTH       METATYPE  ARRAY_LENGTH(*)
----------  -----------  -----------  --------- ------------
ARRAY_TXT   VARCHAR               24  A                 1000
ARRAY_TXT1  VARCHAR               24  A           2147483647
ARRAY_TXT2  VARCHAR               24  L                    9
                                               (*) no underscore on z/OS
```

Note the metatype: A for "ordinary", L for associative
Index type is limited to VARCHAR(n) and INT

Only a few other programming languages support the “associative array” type as a built-in type:

- Perl (they call it a “hash”)
- PHP (every array in PHP is an associative one)
- JavaScript (every object is essentially an associative array with text keys)
- REXX: compound variables (dot notation) are essentially associative arrays!
- also Oracle (PL/SQL) knows both ordinary arrays (VARRAY) and associative arrays

Other languages often support similar constructs, but not really as “built-in” datatypes:

- Python (where it’s called a “mapping”)
- In Java, C# and C++, templated types can be declared which are similar to associative arrays:
    - in C#, associative arrays are called Dictionaries
    - in C++ and Java, they are called maps

  These structures are not really built-ins of these languages
- even Windows PowerShell uses associative arrays
- also, recent versions of bash and ksh support associative arrays

Associative arrays: how to use them

- Declare an SQL PL *variable* to be of associative array type:

```
DECLARE my_list array_txt2;
```

- Filling an array: element-wise, or "as a whole" (constructor):

```
SET my_list['key'] = 'value';  -- any key of max. 9 chars
SET my_list['key'] = my_list['other'] ;  -- overwrites!
```

- The "ARRAY[val1, val2, ...]" and "ARRAY[fullselect]" constructs cannot be used with associative arrays!
- associative array effectively is a two-column list!
  (but with a guarantee of uniqueness for first "key" field)
- useful predicate: test whether a "key" has been used:

```
IF  array_exists(my_list, 'key')  THEN ... END IF;
```

Assignment to associative arrays:

- assignment to an *element* a[key] of an array: identical to assignment to an “ordinary” variable:
    * with the SET statement
    * with SELECT ... INTO a[key] ...
    * by passing a[key] as OUT or INOUT parameter in a procedure call
- assignment as a whole is only possible with the SET statement,
  copying one associative array to an other one.

3. Arrays: interesting use cases: 3.1. result table into array

- SELECT .. INTO array_variable:
  - not possible to directly use an array variable as argument of an INTO
    => use the syntax SET array_variable = ARRAY [fullselect] instead
  - or use the **ARRAY_AGG** aggregate function:

```
CREATE TYPE phonelist AS VARCHAR(24) ARRAY[];
DECLARE numbers phonelist;
SELECT ARRAY_AGG(phone_nr ORDER BY priority)
INTO   numbers
FROM   employee_phones WHERE emp_id = my_id;
```

A very important (new) aggregate function in the context of arrays: ARRAY_AGG(expr ORDER BY ...)

3. Arrays: interesting use cases:
3.2. array into table expression

- Conversely, an array can be turned into a table:
  ```
  SELECT x
  FROM   UNNEST(array_var) AS t(x)
  ```
  Actually, more like a nested table expression
  But filled from "outside DB2", from an in-memory list
  Limitation: 1-column NTE!
- array_var can be either ordinary or associative
- when associative: can return a two-column table (keys & vals):
  ```
  SELECT t.key, t.val
  FROM   UNNEST(array_var) WITH ORDINALITY AS t(key,val)
  ```
- multiple arrays can be "combined into" a single multi-col table

A very important (new) table function in the context of arrays: UNNEST(array)

Note that the UNNEST function with an associative array can return either just the values (1 column)
or both keys and values (i.e., a two-column table):

UNNEST (assoc_array_var) => returns just the values
UNNEST (assoc_array_var) WITH ORDINALITY => keys into first column, values into second column

When UNNEST is passed multiple arguments, arrays of potentially different sizes, a table is returned with
as many columns as the numbr of arrays passed in.
The table size is the size of the largest array. “Missing” elements (from the shorter arrays) are filled with
NULLs.
This resembles an OUTER JOIN of the arrays if we thing of the integer indexes as the join keys.
Note that in this case all array arguments must be ordinary (non-associative).

3. Arrays: interesting use cases:
3.3. alternative to cursors

Probably the most useful use case:

- Alternative for a stored procedure *returning a cursor* ...

```
CREATE PROCEDURE old_style(IN ...)
DYNAMIC RESULT SETS 1
BEGIN DECLARE c CURSOR WITH RETURN FOR   SELECT col FROM ... ;
      OPEN c;
END
```

- ... is a stored procedure *returning an array*:

```
CREATE PROCEDURE new_style(IN .., OUT array_txt returned_list)
BEGIN SET returned_list = ARRAY [ SELECT col FROM ... ];
END
```

Beware: an array is an *in-memory* entity!

The main differences between these two forms:

- in the array case, the procedure first loads the full result set in memory,
  then sends that (big) chunk of data to the caller

  in the cursor situation, it's the caller who will have to run through (fetch) the result set,
  meaning that the connection with the database engine is kept for a longer time
- the caller must be able to declare a parameter variable of type "dynamic array"
  * no need anymore for a cursor type object (simplifies the interface)
  * but caller must support the dynamic array datatype
    => in practice, currrently, only SQL PL & Java are supported !!

Remember also the second SQL PL example, where a user defined function was returning a string concate-
nation (comma-separated) of product names. This function could now be rewritten to return an array of
product names, leaving it to the caller how to use that list!

3.4 ARRAY manipulations

The following scalar function manipulates (modifies) an array:

```
ARRAY_DELETE(array_var)          -- empties the array
ARRAY_DELETE(array_var, idx)    -- removes element at index idx
ARRAY_DELETE(array_var, i1, i2) -- removes a range of elements
```

The latter two forms are only available for associative arrays

All three return an array containing the deleted entries.

The following functions return elements from an array:

```
ARRAY_FIRST(array_var) -- the index [1], or the “smallest” one
ARRAY_LAST(array_var)  -- the “highest” index value
ARRAY_NEXT(array_var, idx) -- the next one (after idx_val)
ARRAY_PRIOR(array_var, idx_val) -- prev one (or NULL)
TRIM_ARRAY(array_var, n) -- returns elements 1 => n-1 as array
```

These (new) array functions are reminiscent of manipulations on **scrollable cursors**!
There is indeed an interesting similarity between the cursor area of a static scrollable cursor and an array,
but with two important differences:

- the content of the scrollable cursor area can only be the result set of a single SELECT statement
  while arrays can be filled at will, and modified, even enlarged, when wanted.
- an array is limited to one column only (or essentially two columns, one without duplicates, in the case of an associative array)
  while a cursor row can have any number of fields, of any built-in datatype

4. The ROW datatype (DB2 for LUW)

- An array can be scrolled through / iterated
  => cf static scrollable cursor!
- *But* lacks the flexibility of a cursor: just one field, not a “record”
- The ROW datatype allows to define a “record” structure
  - only in SQL PL context
  - unfortunately not in DB2 for z/OS
  - special case of so-called “anchored type” (see 2nd example)

```
DECLARE TYPE my_rowtype AS ROW(a INT, b VARCHAR(16), c DATE);
DECLARE my_row1  my_rowtype;
DECLARE my_row2  ANCHOR TO ROW OF table_name; /* or view/cursor */
```

- ARRAYs can be based on a ROW datatype! => **2-dim. structure**

```
DECLARE TYPE my_array AS my_rowtype ARRAY [];
```

The ROW datatype seems to be the “missing link” between (scrollable) cursor (area)s and arrays:
- defines a record structure
- similar to a (temporary) table structure!
  => fields with a name and a data type
- cf COBOL variables / structures (but limited to one level)

Note the “DECLARE TYPE” in the example: alternatively “CREATE TYPE” could have been used here.
The latter would create a globally visible new type definition (in the catalog) while the former is a local definition. Only DB2 for LUW supports the “DECLARE TYPE” statement; this construction is also available for the ARRAY type definitions (but not in DB2 for z/OS).

Note: DB2 for z/OS also knows the ROW concept, but only in the context of the UNPACK function.

Part III: Scenario's, Usage considerations, Nice to Haves

1. Scenario's
   - Passing arrays as routine parameters
   - Interesting use cases
   - Performance aspects
   - Caveats
2. Usage considerations
   - Usage restrictions
   - Work-arounds
3. Nice to haves
   - What Oracle provides
   - What COBOL would like to see

“Table of contents” for part 3.

We are still missing the connection of all this with the “outer world”, that is: letting client applications pass arrays from and to DB2.

In this last part, we’ll mention

- how this can be done
- what are the limitations with the current implementation
- when one should consider doing this (because alternatives are less performant and/or too cumbersome)
- what we are still missing (and will hopefully see in a next DB2 release ;-)

1. Scenarios.
- Passing arrays as routine parameters

Arrays can be passed in & out by the caller of a stored procedure:

```
CREATE TYPE int_array   AS   int  ARRAY  [ ] @
CREATE PROCEDURE add_prods_to_order(IN     ordno   INT,
                                    INOUT  prodno  int_array)
BEGIN
  DECLARE detailno SMALLINT;
  DECLARE idx      SMALLINT DEFAULT 1;
  IF 0 = (SELECT count(*) FROM orders WHERE orid = ordno) THEN
    INSERT INTO orders(orid) VALUES (ordno);
  END IF;
  SELECT max(odno)+1 INTO detailno FROM orderdetails WHERE od_orid = ordno;
  WHILE prodno[idx] IS NOT NULL DO
    INSERT INTO orderdetails (od_orid, odno, od_prid)
      VALUES (ordno, detailno, prodno[idx]);
    SET idx = idx + 1;
    SET detailno = detailno + 1;
  END WHILE;
END @
CALL add_prods_to_order(27, my_int_array) @
```

A stored procedure or a user-defined scalar function can be specified to accept or return arrays.
This example (which extends the very first example of this presentation) shows how this could be done.
But of course... the more important question is: what should the *caller* of such a routine do?
(Answer on the next slide)

1. Scenarios.
- Passing arrays as routine parameters (cont.d)

But ... "my_int_array" can only be declared & used in SQL PL ?!

```
DECLARE my_int_array int_array;
SET my_int_array[1] = ...; SET my_int_array[2] = ...; ...
CALL add_prods_to_order(27, my_int_array);
```

=> would be of limited use to "typical" DB2 client programs!

Currently, only **3 other contexts** support calling such procedures:

- A *Java* program using **JDBC**

```
int[] prods=new int[3]; prods[0]=500; prods[1]=600; prods[2]=200;
CallableStatement stmt=conn.prepareCall("CALL add_prods_to_order(27,?)");
stmt.setArray(0, conn.createArrayOf("INT", prods));
```

- The DB2 for LUW command-line processor (**CLP**) also towards DB2 for z/OS

```
db2 "CALL add_prods_to_order(ARRAY[500,600,200])"
```

- the **Data Studio** GUI (also written in Java ;-)

  specify the input array parameter as a *comma-separated list* in the GUI dialog box:

```
500,600,200
```

It turns out that the possibilities are rather limited!
=> For efficiency reasons, the caller must support dynamic in-memory ARRAYs of DB2 datatype elements
- this excludes STATIC SQL (COBOL, C, SQLJ) => arrays would be fixed-size ...
- currently, only JDBC is supported !!
plus (as special cases, since also using JDBC):
- Data Studio
- CLI programs with DB2 for LUW (also when connected to DB2 for z/OS)
- the DB2 CLP of Linux / Unix / Windows
=> More specifically: the Java JDBC connection object provides a new method `createArrayOf`
which converts between DB2 arrays and Java arrays
- note the different indexing: Java array indexes start from 0, not 1

Note that the JDBC syntax is valid both for IN and OUT parameters (and even for INOUT parameters)
With an OUT parameter, the CLP just displays the content of the array as a CSV (vertically aligned),
almost as the result of a SELECT statement with one column (apart from the commas)
Similarly, the Data Studio GUI will show the content of a returned OUT or INOUT parameter as a vertical list

Also note that only “ordinary” SQL PL arrays are supported in this way, not associative arrays!

Some ideas to bypass these limitations are proposed in section 2 of part III.

1. Scenarios.
- Interesting use cases

The main “goal” of passing array parameters is ***call efficiency***!

- in / out param is bundeled into *single* in-memory data “blob”

  as opposed to e.g. returning a **result set** (= opened cursor)

- partial flexibility is kept:
  - array is of variable size
  - (only on DB2 for LUW) can contain ROWs
  - in DB2 for z/OS, it’s like returning a single-column result set
  - careful with *large arrays*, since they are passed in-memory!

Some *use cases*:

- **cache** some intermediate (DB2) data for use in multiple places

  esp. when generating / computing this data is *expensive*

- efficient (but limited) alternative to **multi-row** fetches & inserts

Caching data for multiple use is clearly more performant than computing it (or letting DB2 run a complicated query) multiple times.

That's exactly the most often used argument for using static scrollable cursors.

Only: the latter can only store the result set of a single query.

In-memory arrays are more flexible in that respect, and are stored client-side instead of in a cursor area (which could also be an in-memory cache, but stored DB2-side)

Cached data could e.g. be:

- configuration parameters, to be fetched once from DB2 tables, then used multiple times
- complicated client-side computations, resulting in multiple similarly typed results, stored under a single name.
- a "common table expression", but which should be available in several different SQL statements

Unfortunately, on DB2 for z/OS, arrays cannot be stored in so-called session global variables;

on DB2 for LUW, such variables can be of array (or row) type

=> allows for sharing arrays between different SQL PL routines called by the same session

2. Usage considerations.
- Work-arounds for environments other than JDBC

- **COBOL**: will not be able to call such a stored procedure directly
- work-around:
  - create an additional stored procedure calling the first one
  - converting the array elements to a fixed-width (binary) concatenation

    of type VARCHAR(32700), since the array length is variable
  - and having an additional parameter for the size of the array
  - from COBOL, call that procedure, passing a TABLE host variable
  - don’t forget to pass the effective TABLE size (for an IN param) or read it (for an OUT param) and use it in the DEPENDING clause
- **PHP** or other distributed DB2 client environments:
  - use the CLP through a shell interface (not very nice, though...)

REXX: a similar work-around as for COBOL could be set up...

This work-around for COBOL is relatively cumbersome, but has to be done only once.
The COBOL interface is relatively readable, and uses a “cobol array” (called a TABLE) as host variable.
Details are not written out here -- a working example can be obtained from the authors on simple request.

Alternatively, certainly for environments that have XML support, an intermediate stored procedure could be created which converts arrays from/to XML objects. In this process, though, the original DB2 data types are lost since XML is intrinsically textual. Also the performance of such a solution would be bad...

2. Usage considerations.
- Work-arounds for environments other than JDBC

- Generic work-around for a single array OUT parameter:

  Example: suppose the following stored procedure must be called:

  ```
  prods_of_order(IN ordno INT, OUT int_array)
  ```

  Create an intermediate table function:

  ```
  CREATE FUNCTION tbl_prods_of_order(ordno  INT)
  RETURNS TABLE (product INT)
  BEGIN ATOMIC
     DECLARE prod_list int_array;
     CALL prods_of_order(ordno, prod_list);
     RETURN
        SELECT prodno
        FROM   UNNEST(prod_list) AS p(prodno);
  END
  ```

  unfortunately, not yet possible in DB2 11 for z/OS ...

- Alternative: intermediate stored proc with result sets ...

  but... cumbersome caller interface in e.g. COBOL which we wanted to avoid!

This work-around looks perfect: it returns the array though a cursor !
Only ... it is limited to
- stored procedures with just a single ARRAY parameter, which moreover is an OUT parameter
- DB2 for LUW, since DB2 11 for z/OS does not yet support multi-statement table functions in SQL PL
- alternatively, one could implement such a table function in Java ... compiled in z/OS, of course!

Second alternative would be the “classical” approach, without any limitations:
- multiple output “arrays” possible
- the “arrays” could even consist of “rows”, even on DB2 for z/OS

Well, instead of arrays we would just use so-called “result sets”, i.e., opened cursors returned to the caller
Now, this was exactly what we wanted to avoid in the first place, when introducing arrays!

3. “Nice to have”s

- The ARRAY implementation of DB2 for LUW (since 9.5) is fairly complete! (e.g. including ARRAYs of ROWs)

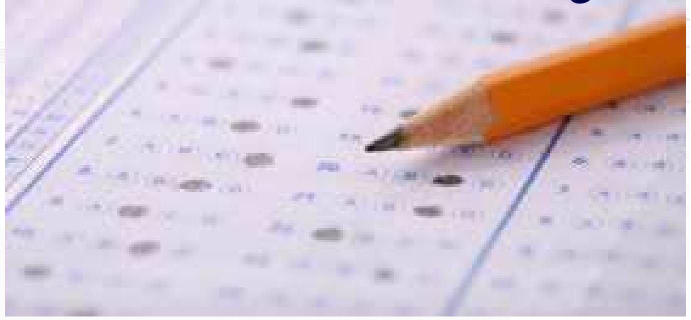
  => first nice-to-have: similar functionality in DB2 for z/OS
  - including ROWs as anchored types to tables, views & especially also *cursors*
  - including the DECLARE TYPE possibility in SQL PL
- Ideas from Oracle PL/SQL:
  - support “tabular” arrays (nested tables), including support for UNION, JOIN, ...
  - INSERT INTO <table-name> VALUES <row-var> & FETCH c INTO <row-var>
- Ideas from COBOL:
  - pass nested structure of DATA DIVISION vars to/from DB2 *keeping their structure* (maybe need ROWs from ARRAYs from ROWs from ... in DB2 ?)
- Built-in function to compare arrays
  - could e.g. be used for comparing two (small) tables

Some random ideas that could pop up in some practical situation, and then turn out to be missing
Especially PL/SQL (Oracle) has an interesting alternative view on ARRAYs
- they use the terms VARRAY, associative array, and nested table
- all of these essentially correspond to DB2 for LUW's ARRAY of datatype ROW
  => with ordinary ARRAY of built-in type, and ROW (single-element array) as special cases
- especially the INSERT ... VALUES <varray> syntax is a "must-have"
  => note how similar this is to what the COBOL (precompiler) allows!

Peter Vanroose
ABIS Training & Consulting

pvanroose@abis.be

Please fill out your session evaluation before leaving!



Session: F4

Title: New SQL PL data types - with use cases you probably did not think of!

Thanks for attending / listening / reading !