Native Stored Procedures Best Practices

Native Stored Procedures Best Practices

What is a Native Stored Procedure? They are simply packages, with "runtime structures" for the SQL statements to be executed. They are simply a set of SQL statements (some SQL procedure processing code, and some SQL DML statements). They are called just like any external stored procedure, where the call comes into the DBM1 address space. Because native stored procedures are DB2 objects just like the external procedures, they get defined to the DB2 catalog and then the integrity of the parameters to be passed is protected and checked. The source code is written entirely in SQL PL, with the program logic being part of the stored procedure definition (within the CREATE PROCEDURE statement itself). Also part of the CREATE PROCEDURE is the Bind parameters needed to go along with the generated package.

DB2 finds the package, matches up the 'Call' to the stored procedure definition, loads the package and executes the statements. They do not use below the bar storage, and are not assigned a WLM to run under.

No external load module is created for native SQL language procedures. The entire executable is contained within the package in DB2. When you create a native SQL stored procedure, the procedural SQL statements are stored in the DB2 catalog and directory, as are the SQL statements that are used for accessing your DB2 data. They run in the DBM1 address space, so there is no need to create a WLM environment to manage the procedures. As a result, when you prepare a native SQL procedure, the entire executable is contained within DB2. This simplifies the deploy process since you don't have to worry about code level management in load libraries and in WLM application environments.

In contrast, an external stored procedure with SQL needs a complete language environment for the user program, and an assigned WLM to run under. The external program comes back to DBM1 to get its package loaded and SQL statements executed. This is what gets "throttled" (the external program execution environments and their associated TCBs). When an incoming stored procedure request is queued for WLM, the DB2 thread is suspended in DBM1. Many customers have experienced delays and DBM1 storage problems when their WLM goals weren't adjusted properly and the queued requests built up. The solution is to either adjust the WLM goals, or else adjust the limit on DB2 threads (local and/or distributed). For external stored procedures, the stored procedure definition and the program logic are two separate components with the Bind parameters being part of the external programs package generation.

© 2017 Themis, Inc. All rights reserved.

DRDA® activity is a candidate for zIIP rerouting. Remote native SQL procedures being called through a distributed thread run under an enclave's SRB instead of a TCB in the DBM1 address space are candidates for zIIP rerouting with DB2 V9. The zIIP is a specialty engine (System z9™ Integrated Information Processor) and if one is available, z/OS will manage and direct work between the general purpose processor (the portion of the mainframe that traditionally has handled the z/OS workload) and the zIIP specialty engine(s). Work that runs on the zIIP does not incur software charges and factor into software pricing based on the service units consumed; therefore it is a very attractive lower-cost alternative to running workloads on a general purpose processor. Many customers have gone to native stored procedures specifically for this reason. This also reduces the workload on the Central Processor so additional work can be run if needed.

Information about external SQL language and native SQL language stored procedures is contained in the following tables:

- SYSIBM.SYSENVIRONMENT
- SYSIBM.SYSROUTINES_OPTS
- SYSIBM.SYSROUTINES_SRC
- SYSIBM.SYSROUTINESTEXT
- SYSIBM.SYSROUTINESAUTH

© 2017 Themis, Inc. All rights reserved.

Advantages of Native Stored Procedures:

- **Simplified Build / Deploy process:** No external address space environment needed and no compilers. The IBM Data Studio tool provides a great editor window and deployment for the SQL PL language, making it very easy to code, debug, test, and deploy.
- **Better Performance (at times):** It is not a guarantee that a native stores procedure will always outperform an external stored procedure. But the fact that it is run entirely in the DB2 engine versus an external address space and an assigned WLM environment gives it a boost. The SQL PL language is a procedural language that is converted to byte code and stored in the package. At run time it is then compiled further and executed. This is not always as efficient as compiled 'C', Cobol, etc. Stored procedures with a lot of processing code may not always be as efficient. This process has improved some in V10. V10 has a REGENERATE statement that is part of the ALTER that will not only REBIND the SQL DML statements for possible better access paths, but also regenerate the SQL Control statements for better efficiency.
- **zIIP eligible:** Native SQL procedures are eligible for offloading to a zIIP. If a native SQL procedure is called from a DRDA client using TCP/IP, then a portion of the SQL procedure processing is directed to a zIIP.
- **Compatibility:** z/OS offers SQL PL language that is now more compatible with DB2 LUW, and other platforms.
- **Easy to Learn:** The SQL PL language is an easy language to learn for any developer.

SP Naming/Authorization: A native stored procedure is like any other DB2 object, where its name consists of 2 parts (an owner/schema and the SP name). Native stored procedures by default will take on the owner/schema of the Collection ID it gets bounds into, that is the collection ID in the bind process will be the same as the schema name defined in the Create Procedure. For native stored procedures, the bind parameters are now a part of the SP header information because in the deployment, the code will be bound into a package.

© 2017 Themis, Inc. All rights reserved.

1) Typically, it's best to put a native stored procedure in a Collection ID that matches the owner/schema of the tables being processed. For example: if the tables being processed are all under STESTID, then the stored procedure should be bound into a collection named STESTID which happens automatically if the stored procedure gets named and deployed as STESTID.SPNAME. This often goes against the normal DBA naming conventions for collections, but makes it easy for developers to code unqualified stored procedure calls, and have the call fall under the assigned qualifier specified in their own bind parameters.

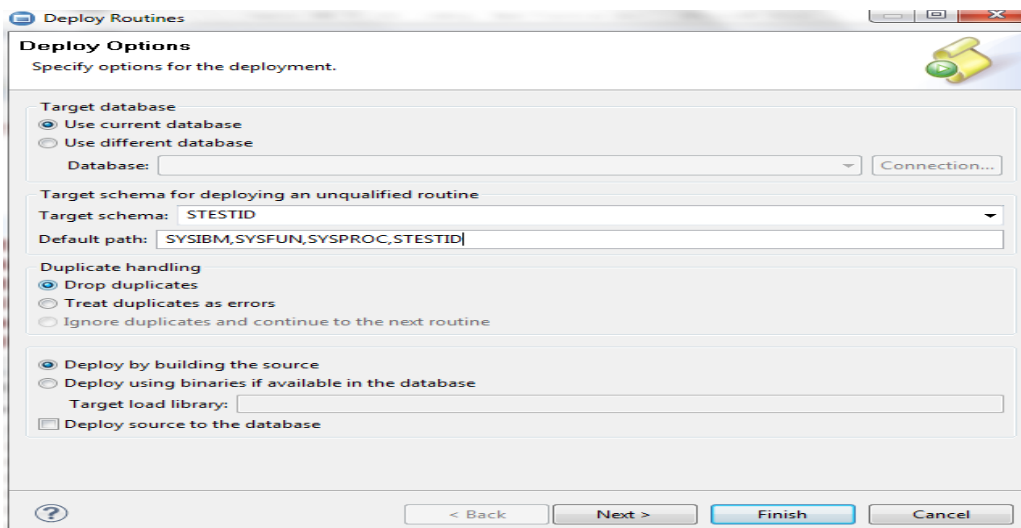
   You may have to execute the following:

   'GRANT CREATEIN ON SCHEMA STESTID TO XXXXXXXX'.

   The CREATEIN privilege is always required to create a stored procedure in a given schema. By executing the CREATE on SCHEMA, this allows

   Since there may be many application developers or DBAs creating native stored procedures into the same schema, you may want to grant the CREATEIN privilege on the schema to a secondary authid that represents a group of users who create stored procedures. Each application developer could then issue a SET CURRENT SQLID statement to the secondary authid, or set it in Data Studio as part of the deploy wizard.

   In the IBM Data Studio tool, the deploy wizard asks to pick a target schema. Whatever is in this schema name will become the collection ID the package is created in. The following screen shot states target schema as STESTID which becomes the collection ID.



© 2017 Themis, Inc. All rights reserved.

2) Note that in Data Studio, the Target Schema specified in the deploy wizard controls both the owner of the stored procedure and the collection Id where the stored procedure is bound into, unless PACKAGE OWNER is specified in the Create Procedure header. When this is stated then the target schema only controls the collection ID. If secondary authids are being used, then that secondary authid needs to have the appropriate privileges, and the developer creating the stored procedure should have PACKAGE OWNER XXXXXXXX where XXXXXXXX contains the secondary authid.

3) If the stored procedure being created contains SQL statements, a package will be created and stored in the DB2 catalog. The BINDADD privilege is required to create new packages in a DB2 subsystem. The SQL to grant BINDADD privilege to authid PGRM123 is as follows: GRANT BINDADD TO 'PGRM123'.

   The authorization ID used to create the native stored procedure package must have the following privileges. If a secondary auth ID is being used, then that ID is the one needing these privileges.

   - BINDADD
   - CREATE IN COLLECTION collection
   - CREATEIN ON SCHEMA schema

4) There must also be an EXECUTE privilege on the stored procedure.

5) Native stored procedures do not need to have their collection bound to a plan, unless the stored procedure is being called from packages that execute under a plan. Typically native stored procedures are first written to be called from distributed threads where binding the Collection ID to a plan does not come into play.

© 2017 Themis, Inc. All rights reserved.

SQL PL Header: The first part of a SQL PL stored procedure (the header) contains information about the parameters to be passed to and from the procedure, as well as additional options that control the behavior of the procedure. And since the procedure will be bound into a package, the header area will also contain its bind options for the package. The created package will be larger than a typical package in that it contains both the optimized SQL and the procedure processing byte code.

Parameters:

- Inout Parameters: Only the parameters defined as Inout can both programs set and the other program see.
- Input Parameters: Input only to the stored procedure. If the stored procedure program overlays the input parameter, it is not seen from the calling program.
- Output Parameters: These parameters all always received as null to the stored procedure (no matter what the calling programs initializes them to), and thus should be initialized first thing in the processing code of the stored procedure code.

I like the idea of having parameters follow naming rules:

- All begin with either P_IN, P_INOUT, or P_OUT

SQL PL Body: This is the program processing code, and the order in which the SQL Body is laid out is extremely important. If any of the following are out of physical order, then confusing syntax errors occur in the deploy process. This does not mean that processing code has to have something defined in each of these areas, but if there exist code for any of these areas they must be in the following order.

1) Declare Variables
2) Declare any Conditions
3) Declare Cursors
4) Declare Error Handlers
5) Processing Code

In SQL PL, a variable is a meaningful name of a temporary storage location that supports a particular data type in the program. In order to use a variable, you need to declare it in top of the SQL PL block before any error handling routines or declare cursors reference them.

© 2017 Themis, Inc. All rights reserved.

SQL PL Variable Naming Convention: Like other programming languages, a variable in SQL PL must follow the naming rules as below. I particularly like to have all variable names begin with V_.

- The variable name may be up to 128 characters. Try to make it as meaningful as possible.
- The starting of a variable must be an ASCII letter. It can be either lowercase or uppercase. Note that SQL PL is not case-sensitive.
- A variable name can contain numbers, underscore, and dollar sign characters followed by the first character. Again, do not make your variable names hard to read or understand. Make it easy for others understand and maintain in the future. Dashes are not allowed.
- Variables are only available to the procedure itself.
- Definitions are identical to the manner in which DB2 table columns are defined and must use the same data types.
- All variables must be defined before any error handlers or declare cursors. It is recommended to place them right away in the SQL body (P1: Begin …)
- Variable names can be the same names as columns, but is not recommended. If it is ambiguous to the compiler, it assumes it is a parameter or variable name.
- Variable names are not case sensitive. This means you cannot have a variable V_LASTNAME and one v_lastname.
- Variable names use underscores in them, dashes create errors. For example a declared variable for V-LASTNAME will get an error.
- In the SQL/PL language are variables are considered SQL variables, and thus are not preceded by a colon ':' within the source code.

© 2017 Themis, Inc. All rights reserved.

Nulls

When selecting data into host variables, do not define and include null indicators as part of the SELECT INTO. Null indicators are used in Cobol and some other languages, but not in SQL PL. Select the column directly into the host variable, then check the host variable for NULL.

For example:

```
SELECT MGRNO
 INTO V_MGRNO
 FROM DEPT
 WHERE ……

 IF V_MGRNO IS NULL THEN
    SET V_MGRNO =  ' ';
 ELSE
   SET P_OUT_MGRNO  = V_MGRNO;
 END_IF;
```

The COALESCE function will also work, setting the default of spaces if the value is null coming back from DB2: The VALUE and IFNULL functions also work and do the exact same logic as the COALESCE. The COALESCE is the ANS/ISO standard and it is recommended that it be used. For example:

```
 SELECT COALESCE(MGRNO, ' ')
 INTO V_MGRNO
 FROM DEPT
 WHERE ……


SET P_OUT_MGRNO = V_MGRNO;
```

Rebinds

If native stored procedures packages need rebound with any new / different bind parameters, the 'Create Procedure' DDL will need to be changed also, then rerun.

This is an easy process in the IBM Data Studio tool, where the header information is changed and the Deploy is run again. In this tool, subsequent deploys on an already created stored procedure actually runs a Alter Procedure statement.

© 2017 Themis, Inc. All rights reserved.

Parameter Changes

This is the same as external stored procedures in that if there are any parameter changes (definition changes, adding or deleting another parameter), the stored procedure must be dropped, committed, and then recreated/redeployed.

Checking for Not Found Condition

Should have a specific error handler for not found, as follows.

```
 Declare Continue Handler for not found
Begin
  set v_sqlcode = sqlcode;
End;
```

V_sqlcode has to be one of the declared variables, defined the same as Sqlcode which is integer.

After any SQL statements where a not found is possible, if +100 is returned, the v_sqlcode gets set to +100 because of the continue handler above. This v_sqlcode needs to be checked after the SQL statement, not the actual Sqlcode. The actual Sqlcode of +100 will get reset to 0 after the 'set v_sqlcode = sqlcode' is executed. This statement should always be the first statement in every error handling logic, and SQL PL processing code should always check the v_sqlcode within its logic, and not the sqlcode directly.

Make sure before each statement that could return a sqlcode of +100, that the v_sqlcode is set to 0. It could still have a +100 from some previous statement in the code, and if the sqlcode is 0, then the Not Found continue handler is not executed. For example:

```
SET V_SQLCODE = 0;
  SELECT  MGRNO
  INTO V_MGRNO
  FROM DEPT
  WHERE DEPTNO = V_DEPTNO
     AND LOC       = V_LOC
   WITH UR;

   IF V_SQLCODE = 0  THEN
     SET  ......  ;

        END IF;
```


© 2017 Themis, Inc. All rights reserved.

Error Handlers: Capturing SQLCODE and SQLSTATE:

If the following variables are declared in the code, then they will be populated after each SQL statement executed in the code. Each SQL statement is both a database calls and an SQL PL procedural statements.

```
DECLARE SQLCODE INTEGER  DEFAULT 0;
DECLARE SQLSTATE CHAR(5) DEFAULT '00000';
```

It is not possible to retrieve other information on a call (as in the Cobol SQLCA area being populated) without executing a GET DIAGNOSTICS command. This is the statement used to gather further information based on any returned SQLCODE.

Note: Every SQL PL procedural statement will return a SQLCODE and SQLSTATE.

For example:

```
Set V_ACCT_NUM = 0 ;                Sets the SQLCODE and SQLSTATE.

IF V_ACCT_NUM > 0 then              Sets the SQLCODE and SQLSTATE.
   ..............
      Case
         When V_ACCT_NUM = 0  then ...   Sets the SQLCODE and SQLSTATE

      Fetch into .....                   Sets the SQLCODE and SQLSTATE.
```

In the following example, the P_RETCODE will get set to 0 because the 'IF' check on SQLCODE = +100 resets the SQLCODE back to 0. This is a real SQL PL programming

'Gotcha'.

```
Fetch C1 into V_ACCT_NUM;
IF SQLCODE = +100 then              ← SQLCODE gets reset to 0
   Set P_RETCODE = SQLCODE;         ← P_RETCODE gets set to 0
END IF;
```

© 2017 Themis, Inc. All rights reserved.

Error Handlers: EXIT and CONTINUE:

Exit and Continue handlers are defined to be executed whenever certain errors occur within the procedural code. These are defined at the top of the code after any declared variables and declared cursors. They will contain statements to be executed whenever any errors (database calls or procedural statements) that occur. These are very similar to the CICS Handlers defined in CICS code. These exit and continue handlers get defined once, then anywhere within the processing code an error occurs, processing automatically jumps to one of these routines.

Note: If no error handlers exist, and an SQL error occurs in the stored procedure, the SQL error code is immediately returned to the calling program as an error on the SQL CALL. This can be misleading because the CALL was actually good yet an error occurred within the stored procedure.

Exit handlers are defined as either EXIT or CONTINUE. That means when the handler is automatically initiated due to an error, the code within the handler will execute and then either EXIT back to the calling program or CONTINUE after the statement where the error occurred. For example: SQLEXCEPTION is for any negative SQLCODE error.

```
DECLARE V_SQLMSG          VARCHAR(250) DEFAULT ' ';
DECLARE V_LINE_NUM        INTEGER DEFAULT 0;
DECLARE V_REASON_CODE  INTEGER DEFAULT 0;
DECLARE V_SQLCODE_OUT  INTEGER DEFAULT 0;


DECLARE EXIT HANDLER FOR SQLEXCEPTION
  BEGIN

        GET DIAGNOSTICS CONDITION 1
          V_SQLMSG            = MESSAGE_TEXT
        , V_LINE_NUM          = DB2_LINE_NUMBER
        , V_REASON_CODE    = DB2_REASON_CODE
        , V_SQLCODE_OUT    = DB2_RETURNED_SQLCODE;

        SET P_RETCODE = SQLCODE;

  END;


   DECLARE CONTINUE HANDLER FOR NOT FOUND
     BEGIN
        SET V_RETCODE = SQLCODE;
     END;
```


© 2017 Themis, Inc. All rights reserved.

Exit or Continue Handlers can be defined for the following conditions.

SQLEXCEPTION: Any negative SQLCODEs that occur.
SQLWARNING : Any positive SQLCODEs that occur (except +100 Not Found).
NOT FOUND : SQLCODE = +100.

or

For specific handlers. The associated SQLSTATE must be specified. For example if you want to capture a duplicate insert (SQLCODE = -803), you would code:

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '23505'

  BEGIN
   SET V_RETCODE = SQLCODE;
  END;
```

Note: For an SQLCODE = -803, the code will jump to this continue handler, and all other negative SQLCODEs will jump to the SQLEXCEPTION handler.

A +100 will not jump to a defined SQLWARNING handler.

Get Diagnostics: This statement is needed to obtain further information on an error because the only 2 pieces of error information that automatically gets sent from DB2 are the SQLCODE and SQLSTATE. A declared variable must be defined for DB2 to place the additional information into. Following are the definitions for all the different pieces of information that can be retrieved by calling the Get Diagnostics. This statement will return diagnostic information about the last SQL statement that was executed, and return information about the following:

Statement: Diagnostic information about the statement as a whole (Condition 1)

Condition Items: This would be for condition items other than 1, and is used with multi row processing. This can contain information about individual errors that occurred within a multi row processing statement.

Connection: This contains information about the SQL statement if it was a connect statement.

© 2017 Themis, Inc. All rights reserved.

The most common diagnostic informational items would be:

- MESSAGE_TEXT: Message text from statement in error. VARCHAR(250) should be sufficient.
- DB2_RETURNED_SQLCODE: SQLCODE from statement in error.
- RETURNED_SQLSTATE: SQLSTATE from statement in error.
- DB2_LINE_NUMBER: Line number of statement within SQL PL code.
- DB2_REASON_CODE: Reason code for the SQLCODE statement in error.

The following are diagnostic informational items that match up to what is returned in the SQLCA area of Cobol programs.

- DB2_MODULE_DETECTING_ERROR.
- DB2_SQLERRD_SET.
- DB2_SQLERRD1.
- DB2_SQLERRD2.
- DB2_SQLERRD3.
- DB2_SQLERRD4.
- DB2_SQLERRD5.
- DB2_SQLERRD6.

Here is the link to the Get Diagnostics description and definitions:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp?topic=%2Fdb2%2Frbafzmstgetdiag.htm

© 2017 Themis, Inc. All rights reserved.

Table 55. Data Types for GET DIAGNOSTICS Items

Item Name	**Data Type**
Statement Information Item	
COMMAND_FUNCTION	VARCHAR(128)
COMMAND_FUNCTION_CODE	INTEGER
DB2_DIAGNOSTIC_CONVERSION_ERROR	INTEGER
DB2_GET_DIAGNOSTICS_DIAGNOSTICS	VARCHAR(32740)
DB2_LAST_ROW	INTEGER
DB2_NUMBER_CONNETIONS	INTEGER
DB2_NUMBER_PARAMETER_MARKERS	INTEGER
DB2_NUMBER_RESULT_SETS	INTEGER
DB2_NUMBER_ROWS	DECIMAL(31,0)
DB2_NUMBER_SUCCESSFUL_SUBSTMTS	INTEGER
DB2_RELATIVE_COST_ESTIMATE	INTEGER
DB2_RETURN_STATUS	INTEGER
DB2_ROW_COUNT_SECONDARY	DECIMAL(31,0)
DB2_ROW_LENGTH	INTEGER
DB2_SQL_ATTR_CONCURRENCY	CHAR(1)
DB2_SQL_ATTR_CURSOR_CAPABILITY	CHAR(1)
DB2_SQL_ATTR_CURSOR_HOLD	CHAR(1)
DB2_SQL_ATTR_CURSOR_ROWSET	CHAR(1)
DB2_SQL_ATTR_CURSOR_SCROLLABLE	CHAR(1)
DB2_SQL_ATTR_CURSOR_SENSITIVITY	CHAR(1)
DB2_SQL_ATTR_CURSOR_TYPE	CHAR(1)
DYNAMIC_FUNCTION	VARCHAR(128)
DYNAMIC_FUNCTION_CODE	INTEGER
MORE	CHAR(1)
NUMBER	INTEGER
ROW_COUNT	DECIMAL(31,0)
TRANSACTION_ACTIVE	INTEGER
TRANSACTIONS_COMMITTED	INTEGER
TRANSACTIONS_ROLLED_BACK	INTEGER

© 2017 Themis, Inc. All rights reserved.

Table 55. Data Types for GET DIAGNOSTICS Items

Item Name	Data Type
Connection Information Item	
CONNECTION_NAME	VARCHAR(128)
DB2_AUTHENTICATION_TYPE	CHAR(1)
DB2_AUTHORIZATION_ID	VARCHAR(128)
DB2_CONNECTION_METHOD	CHAR(1)
DB2_CONNECTION_NUMBER	INTEGER
DB2_CONNECTION_STATE	INTEGER
DB2_CONNECTION_STATUS	INTEGER
DB2_CONNECTION_TYPE	SMALLINT
DB2_DYN_QUERY_MGMT	INTEGER
DB2_ENCRYPTION_TYPE	CHAR(1)
DB2_PRODUCT_ID	VARCHAR(8)
DB2_SERVER_CLASS_NAME	VARCHAR(128)
DB2_SERVER_NAME	VARCHAR(128)
Condition Information Item	
CATALOG_NAME	VARCHAR(128)
CLASS_ORIGIN	VARCHAR(128)
COLUMN_NAME	VARCHAR(128)
CONDITION_IDENTIFIER	VARCHAR(128)
CONDITION_NUMBER	INTEGER
CONSTRAINT_CATALOG	VARCHAR(128)
CONSTRAINT_NAME	VARCHAR(128)
CONSTRAINT_SCHEMA	VARCHAR(128)
CURSOR_NAME	VARCHAR(128)
DB2_ERROR_CODE1	INTEGER
DB2_ERROR_CODE2	INTEGER
DB2_ERROR_CODE3	INTEGER
DB2_ERROR_CODE4	INTEGER
DB2_INTERNAL_ERROR_POINTER	INTEGER
DB2_LINE_NUMBER	INTEGER
DB2_MESSAGE_ID	CHAR(10)

© 2017 Themis, Inc. All rights reserved.

Table 55. Data Types for GET DIAGNOSTICS Items

Item Name	Data Type
DB2_MESSAGE_ID1	VARCHAR(7)
DB2_MESSAGE_ID2	VARCHAR(7)
DB2_MESSAGE_KEY	INTEGER
DB2_MODULE_DETECTING_ERROR	VARCHAR(128)
DB2_NUMBER_FAILING_STATEMENTS	INTEGER
DB2_OFFSET	INTEGER
DB2_ORDINAL_TOKEN_n	VARCHAR(32740)
DB2_PARTITION_NUMBER	INTEGER
DB2_REASON_CODE	INTEGER
DB2_RETURNED_SQLCODE	INTEGER
DB2_ROW_NUMBER	INTEGER
DB2_SQLERRD_SET	CHAR(1)
DB2_SQLERRD1	INTEGER
DB2_SQLERRD2	INTEGER
DB2_SQLERRD3	INTEGER
DB2_SQLERRD4	INTEGER
DB2_SQLERRD5	INTEGER
DB2_SQLERRD6	INTEGER
DB2_TOKEN_COUNT	INTEGER
DB2_TOKEN_STRING	VARCHAR(70)
MESSAGE_LENGTH	INTEGER
MESSAGE_OCTET_LENGTH	INTEGER
MESSAGE_TEXT	VARCHAR(32740)
PARAMETER_MODE	VARCHAR(5)
PARAMETER_NAME	VARCHAR(128)
PARAMETER_ORDINAL_POSITION	INTEGER
RETURNED_SQLSTATE	CHAR(5)
ROUTINE_CATALOG	VARCHAR(128)
ROUTINE_NAME	VARCHAR(128)
ROUTINE_SCHEMA	VARCHAR(128)
SCHEMA_NAME	VARCHAR(128)

© 2017 Themis, Inc. All rights reserved.

Table 55. Data Types for GET DIAGNOSTICS Items	
Item Name	**Data Type**
SERVER_NAME	VARCHAR(128)
SPECIFIC_NAME	VARCHAR(128)
SUBCLASS_ORIGIN	VARCHAR(128)
TABLE_NAME	VARCHAR(128)
TRIGGER_CATALOG	VARCHAR(128)
TRIGGER_NAME	VARCHAR(128)
TRIGGER_SCHEMA	VARCHAR(128)

Error Processing: When an SQL error occurs within the stored procedure, the error needs to be logged somewhere. Does the stored procedure log the error? Does the calling program log the error? Does it get logged into a DB2 table or somewhere else? Should a commit be done, and which process should manage the commits, the calling program or the stored procedure? This varies from shop to shop, but following are some guidelines.

Error Logging: Logging the error should be done from within the stored procedure, or error information will need to be passed back to the calling program, or both. Some shops have defined output parameters specifically for error processing that gets captured from the stored procedure GET DIAGNOSTICS command. Even though 99.9% of the time the stored procedure executing in production is error free many shops like to have 5 or 6 output parameters for errors to pass back to let the calling program logged the error. Many times this is because a stored procedure may be called from multiple places, with each environment logging errors differently (for example Websphere vs Java vs Cobol batch). Best practices say for the stored procedure to log the error, and then send the SQLCODE in error back as an output parameter to the calling program, or… the stored procedure calls a Cobol stored procedure that displays the error information in the Workload Manager job sysout, and then passes the same error pieces back to the calling program as output parameters. Calling programs should always be checking 2 statuses when they call a stored procedure. Was the SQL call to the stored procedure good and did the store procedure execute OK? The output parameter containing the last SQLCODE tells the calling program whether the stored procedure executed normally, and any output error parameters will have the information needed for the calling program to log .

Where to Log: Errors need to be logged either in a DB2 Error log table, or a VSAM error log file. The consistent process would be for the native stored procedure to call a Cobol stored procedure passing all the error pieces as input parameters. The Cobol stored procedure receives the input parameters, displays the information in the Workload Manager job sysout, and possibly writes the information to a VSAM file. If the native stored procedure logs the information into a DB2 table, then commit and rollback issues come into play for the unit of work being executed. Having the error information written to a VSAM file ensures the error never gets lost no matter what the calling process decides to do (commit or rollback). However if a native stored procedure is used in read only processes, then

© 2017 Themis, Inc. All rights reserved.

logging into a DB2 error table may be acceptable. IT applications will probably want consistency in stored procedure error processing (native or external), so it's best to understand any current processing in your shop.

Commit/Rollback: Typically a stored procure being called is just a part of the unit of work being executed. Because of this, the calling program should be the one to decide when an error occurred whether to commit or rollback. Because some native stored procedures may be called from many different processes, it is a best practice for the native stored procedure to just log the error and return to the caller with an output parameter noting that an error occurred. Every stored procedure (whether native or not) should have an output parameter defined to let the calling program know that it processed as defined, or some error occurred. Typically this is communicated through an output parameter defined as an integer definition with it getting set to the last SQLCODE that occurred in the processing. This way the calling program will know whether the stored procedure processed as defined or not. V11 has an AUTONOMOUS parameter that can be added at the end of the header information that gives the stored procedure autonomy to executing commits and rollbacks without affecting the calling programs unit of work.

Bind / Deploy: For native stored procedures only:

This bind process allows you to take an already existing and tested native stored procedure and deploy (create) it in other environments. This is a Bind process that can be built into any existing change management process in order to promote these from one environment to another. For example:

```
BIND PACKAGE(CHICAGO.DBPROD)
DEPLOY(DBTHM.SP3N) COPYVER(VR2)
ACTION(ADD) QUALIFIER(THEMISPD
```

This Bind Deploy will create a new SP called DBPROD.SP3N VR2 at the Chicago location by copying it from the DBTHM.SP3N VR2 stored procedure. When creating the new stored procedure at the Chicago location, the stored procedures will be created with a qualifier of THEMISPD. This may have not been the same qualifier as in DBTHM.SP3N VR2.

CHICAGO = Location/LPAR

DBPROD = Owner/Schema of newly created/deployed stored procedure

DBTHM.SP3N = Original / Copied from

Qualifier = Qualifier used for the new DBPROD.SP3N stored procedure.

© 2017 Themis, Inc. All rights reserved.