DB2 for i
Stored Procedures

Kent Milligan, kmill@us.ibm.com
IBM STG Lab Services
ISV Enablement – IBM i
DB2 for i Center of Excellence
“New” Wiki for DB2 Enhancements via PTF

• Regularly check (or Subscribe) to the DB2 for i Technology Updates Wiki!
  – Contains details on new PTFs that deliver new DB2 capabilities
  – Examples:
    • FOR SYSTEM NAME clause for controlling short system name for Tables, Views, and Indexes
    • SQL Query Engine 6.1 support for Logical File on FROM clause
    • CONNECT BY 7.1 support for hierarchical queries
    • RUNSQL CL command
  – Wiki URL:
    https://www.ibm.com/developerworks/ibmi/techupdates/db2

• The wiki is part of a IBM i zone in IBM developerWorks
  https://www.ibm.com/developerworks/ibmi/

• New DB2 for i Blog:  http://db2fori.blogspot.com/
What is a Stored Procedure?

• **Just a called program**
  - Called from SQL-based interfaces via SQL CALL statement
  - Program can be written with SQL procedural language or IBM i high-level languages

• **Supports input and output parameters**
  - Result sets

• **Follows security model of IBM i**
  - Enables you to secure your data
  - IBM i adopted authority model can be leveraged

• **Useful for moving host-centric applications to distributed applications**
What is a Stored Procedure?

- Performance savings in distributed computing environments by dramatically reducing the number of flows (requests) to the database engine
  - One request initiates multiple transactions and processes

- Performance improvements further enhanced by the option of returning result sets
Recipe for a Stored Procedure...

1. Create it

   ```sql
   CREATE PROCEDURE total_val (IN Member# CHAR(6),
                               OUT total DECIMAL(12,2))
   LANGUAGE SQL
   BEGIN
       SELECT SUM(curr_balance) INTO total
       FROM accounts
       WHERE account_owner=Member# AND
             account_type IN ('C','S','M')
   END
   ```

2. Call it (from an SQL interface – SQL CALL) over and over

   ```sql
   CALL total_val('123456', :balance)
   ```
Stored Procedures

- **DB2 for i supports two types of stored procedures**
  - EXTERNAL
    - Register high-level language program (RPG, Java, C, etc) as a stored procedure
    - Procedure may or may not use SQL
  - SQL
    - Entire procedure is coded with SQL
    - Based on SQL Standard (PSM)
      - Natural fit for SQL programmers on IBM i
      - Similar to PL/SQL and T-SQL
      - Same language used for SQL Triggers & Functions

- **SQL CREATE PROCEDURE statement used for both types**
Stored Procedures & Functions

• Comparison of Procedures & User-Defined Functions
  – Stored Procedure
    • Similar to high-level language program, facilitate reuse of common business logic or business process
    • Most commonly used to call procedure on another system
    • Invoked with SQL CALL statement by developer

  – User-Defined Function
    • Similar to high-level language program, facilitate reuse of common business logic or business process
    • Most commonly called as each row being processed/selected during execution of an SQL statement
    • Provides ability to augment the built-in SQL functions provided by DB2
    • Invoked from SQL statement by developer
      SELECT myfunc1() FROM mytable WHERE myfunc2(col1)>100
Create Procedure options

- Procedure name + number of parameters make a unique signature
- Up to 1024 parameters can be passed (pre-V5R4 limit is 253)
Create Procedure Options

- **With 7.1, default values can be supplied for parameters**
  - Supported for both SQL & External procedures
  - Minimizes coding changes when adding new parameters to existing stored procedures
  - Example:
    ```sql
    CREATE PROCEDURE proc1 ( IN p1 INT,
                              IN p2 INT DEFAULT 0,
                              IN p3 INT DEFAULT 55) ...
    ```
  - Possible calls:
    ```sql
    CALL proc1(100) /* p1=100, p2=0, p3=55 */
    CALL proc1(150,1) /* p1=150, p2=1, p3=55 */
    CALL proc1(200, p3=>0) /* p1=200, p2=0, p3=0 */
    ```
Create Procedure options

- **LANGUAGE** - Language of procedure (SQL, C, RPG...)
- **SPECIFIC** – unique "short" name for SQL procedures
  - Automatically generated if not specified
  - Can be used to control C program object name
- **FENCED/NOT FENCED** – No effect on procedures
  - NOT FENCED can improve performance of SQL Functions
Create Procedure options

- **DETERMINISTIC** - procedure will always return the same result from successive calls with identical input arguments

- **PROGRAM TYPE** – SUB enables DB2 to create service program object. Program object created by default.
Create Procedure options

- **NEW SAVEPOINT LEVEL** – new savepoint (transaction level) created on entry

  - **OLD SAVEPOINT LEVEL**
  - **NEW SAVEPOINT LEVEL**

- **COMMIT ON RETURN NO**
  - **COMMIT ON RETURN YES**

- **COMMIT ON RETURN YES** - DB2 issues a commit if the procedure successfully returns
  - Work done by both the invoker and procedure is committed
  - Result Set cursors must be declared WITH HOLD to be usable after the commit operation
Create Procedure options

- **RESULT SETS** specifies max number of result sets that can be returned from procedure
  - More on result sets later on...
Create Procedure options

- MODIFIES SQL DATA – Most any SQL statement allowed
- READS SQL DATA – Read Only statements
- CONTAINS SQL – Simple local statements (SET, DECLARE)
- NO SQL – No SQL allowed (external procedures only)
Create Procedure options

- **SET OPTION** - set processing options
  - Naming option (*SQL vs *SYS), sort-sequence, SQL path, debug...
  - Example: SET DBGVIEW=*STMT, USRPRF=*USER

- **Most interesting options for SQL Procedures are:**
  - USRPRF for adopted authority (defaults to *OWNER)
  - DBGVIEW for creating debuggable version of SQL Procedure
    - *SOURCE enables SQL statement-level debug
SQL Stored Procedure Example

CREATE PROCEDURE proc1 (IN Emp# CHAR(4), IN NwLvl INT)
  LANGUAGE SQL Proc1_Src:
  BEGIN

    DECLARE CurLvl INT;
    SELECT edlevel INTO CurLvl FROM emptbl
    WHERE empno=Emp#;

    IF NwLvl > CurLvl THEN
      UPDATE emptbl SET edlevel=NwLvl,
                       salary=salary + (salary*0.05) WHERE empno=Emp#;
    END IF;

  END

  • DB2 for i uses the source to create a C program object with the IBM i pre-compiler and compiler commands
SQL Compound Statement

- **ATOMIC**
  - all statements succeed or are rolled back.
  - COMMIT or ROLLBACK cannot be specified in the procedure
  - must also be created with COMMIT ON RETURN YES

- **NOT ATOMIC** – no guarantee or atomicity
Basic Constructs

- **DECLARE** – define variable. Initialized when procedure is called
  
  DECLARE v_midinit, v_edlevel CHAR(1);
  DECLARE v_ordQuantity INT DEFAULT 0;
  DECLARE v_enddate DATE DEFAULT NULL;
  
  - Uninitialized variables are set to NULL

- **SET** - assigning a value parameter or variable
  
  SET total_salary = emp_salary + emp_commission;
  SET total_salary = NULL;
  SET loc_avgsalary = (SELECT AVG(salary) FROM employees);

- **Comments** - **two types**
  
  - Two consecutive hyphens (---)
  - Bracketed comments (/* ... */)
Basic Constructs

• Call statement - for invoking stored procedures
  • CALL ProcedureName(Parm1, Parm2, etc);
    - Up to 1024 arguments allowed on CALL statement (253 prior to V5R4)
    - A parameter can contain SQL parameter, SQL variable, constant, special register, or NULL
    - Expressions not yet supported as parameters

• Provides a mechanism for accessing system functions and APIs from an SQL Stored Procedure
Conditional Constructs

• IF statement
  IF rating=1 THEN SET price=price * 0.95;
  ELSEIF rating=2 THEN SET price=price * 0.90;
  ELSE SET price=price * 0.80;
  END IF;

• CASE Expression
  – **First form:**
    CASE workdept
      WHEN 'A00' THEN
        UPDATE department
        SET deptname = 'ACCOUNTING';
      WHEN 'B01' THEN
        UPDATE department
        SET deptname = 'SHIPPING';
      ...
      ELSE UPDATE department
        SET deptname = 'UNKNOWN';
      END CASE;

  – **Second form:**
    CASE
      WHEN vardept='A00' THEN
        UPDATE department
        SET deptname = 'ACCOUNTING';
      WHEN vardept='B01' THEN
        UPDATE department
        SET deptname = 'SHIPPING';
      ...
      ELSE UPDATE department
        SET deptname = 'UNKNOWN';
      END CASE;
Looping Constructs

- **FOR statement** - execute a statement for each row of a table

  ```sql
  FOR loopvar AS
    loopcursor CURSOR FOR
      SELECT firstname, middinit, lastname FROM emptbl
    DO
      SET fullname=lastname||', ' || firstname||' ' || middinit;
      INSERT INTO namestbl VALUES( fullname );
  END FOR;
  ```

- **Allows columns in FOR SELECT statement to be accessed directly without host variables**

- **Cursor can be used in WHERE CURRENT OF... operation**
Looping Constructs

• **Other looping constructiosn**
  - LOOP - repeat forever
    • Use LEAVE statement to end loop
  - REPEAT...UNTIL – exit condition checked at end
  - WHILE – exit condition checked on entry
    END LOOP;

• **Loop control statements**
  - LEAVE – leaves/ends loop
  - ITERATE – go to top of loop
Looping examples

**LOOP Example - fetch_loop:**
```
LOOP
  FETCH cursor1 INTO v_firstname, v_lastname;
  IF SQLCODE <> 0 THEN
    LEAVE fetch_loop;
  END IF;
  ...
END LOOP;
```

**REPEAT Example - r_loop:**
```
REPEAT
  FETCH cursor1 INTO v_firstname, v_lastname;
  ...
  UNTIL SQLCODE <> 0
END REPEAT;
```

**WHILE Example - while_loop:**
```
WHILE at_end=0 DO
  FETCH cursor1 INTO v_firstname, v_lastname;
  IF SQLCODE <> 0 THEN
    SET at_end = 1;
  END IF;
  ...
END WHILE;
```

NOTE: Though they look similar, each example works differently!
FEEDBACK & ERROR HANDLING
Feedback & Error Handling

• GET DIAGNOSTICS
  – Retrieve information about last statement executed
    • Row_count, return_status, error status....
  – CURRENT or STACKED
    • CURRENT – statement that was just executed
    • STACKED – statement before error handler was entered
      – Only allowed within error handler
  – Example:
    DECLARE update_counter INTEGER;
    ...
    UPDATE orders SET status='LATE'
    WHERE ship_date < CURRENT DATE;
    GET DIAGNOSTICS update_counter = ROW_COUNT;
    ...

• Example:
Feedback & Error Handling

- **SQLSTATE & SQLCODE** accessed by declaring variables with those names
  - DB2 will automatically update these variables after each statement

- **NOTE:** Every procedural statement is potentially an SQL statement. These variables may need to be saved after every statement

- Sample usage:
  
  ```sql
  DECLARE SQLSTATE CHAR(5);
  DECLARE SQLCODE INTEGER;
  
  DELETE FROM tablex WHERE col1=100;
  IF SQLSTATE='02000' THEN ....
  ```
Feedback & Error Handling

Conditions and Handlers

• CONDITION
  DECLARE condition name CONDITION FOR string constant;
  – Allows alias for cryptic SQLSTATE
  – Condition name must be unique within the Stored Procedure

• HANDLER
  DECLARE type HANDLER FOR condition;
  – Type
    • UNDO - rollback statements in compound statement (must be ATOMIC)
    • CONTINUE – continue processing
    • EXIT – exit compound statement
  – Condition
    • Defined condition (above)
    • SQLSTATE ‘xxyzz’
    • SQLWARNING, NOT FOUND, SQLEXCEPTION
CREATE PROCEDURE proc1()

... 

BEGIN

-- row not found condition
DECLARE row_not_fnd CONDITION FOR '02000';

DECLARE CONTINUE HANDLER FOR row_not_fnd
  SET at_end='Y'; -- set local variable at_end

...

DELETE FROM tablex WHERE hiredate>='04/01/2011';

END
Feedback & Error Handling

- SIGNAL & RESIGNAL should be used to pass back error or status to the invoker
  - **SIGNAL**: SIGNAL *condition info SET assign value*;
    - Condition info – condition name or SQLSTATE ‘xxyzz’
    - SET clause provides ability to pass along additional diagnostic information
      - MESSAGE_TEXT most commonly used
      - Values that can be retrieved via GET DIAGNOSTICS
  - **RESIGNAL**: RESIGNAL [*condition info SET assign value]*;
    - Can be used only within handler
    - Can just RESIGNAL – “bracket” info is options
    - Condition info – condition name or SQLSTATE ‘xxyzz’
    - SET clause provides ability to pass along additional diagnostic information
      - MESSAGE_TEXT most commonly used
      - Values that can be retrieved via GET DIAGNOSTICS

- SIGNAL/RESIGNAL information is copied back to the SQLCA of the stored procedure invoker
  - **EXAMPLE**: VB program could retrieve the SQLSTATE and message text via the Connection object (Conn.Error(i).SQLSTATE & Conn.Error(i).Description)
CREATE PROCEDURE Change_Salary(IN i_empno CHAR(6),
   IN i_change DEC(9,2) )
   SPECIFIC CHGSAL LANGUAGE SQL
BEGIN

DECLARE EXIT HANDLER FOR SQLSTATE '38S01'
   RESIGNAL SQLSTATE '38S01'
      SET MESSAGE_TEXT = 'CHGSAL: Change exceeds limit.';

DECLARE EXIT HANDLER FOR SQLSTATE '02000'
   SIGNAL SQLSTATE '38S02'
      SET MESSAGE_TEXT = 'CHGSAL: Invalid employee nbr.';

-- check, if the new compensation within the limit
IF (i_change > 25000)
   THEN SIGNAL SQLSTATE '38S01';
END IF;

UPDATE employee SET salary=v_salary + i_salary WHERE empno = i_empno;
END
Feedback & Error Handling

- The RETURN statement can be used to communicate high-level success/failure status to the caller:
  - `RETURN <optional integer value>;`
  - If no return statement is specified, then:
    - If SQLCODE is greater than or equal to 0, the return value is set to 0.
    - If SQLCODE is less than 0, the return value is set to -1.

- Accessing the return value:
  - When invoked by another procedure:
    - `GET DIAGNOSTICS statusvar = RETURN_STATUS;`
  - `?=CALL <procedure name>"` syntax common in ODBC and JDBC
  - Returned in SQLERRD[0]
CREATE PROCEDURE ModAgency(IN agencyVID INTEGER,
    IN agencyNUM INTEGER, IN agencyID INTEGER, IN agentNID INTEGER)
...
BEGIN
...
SET CurrentDT = CURRENT TIMESTAMP;

UPDATE agency
  SET agency_vid=agencyVID, agency_num=agencyNUM,
      agent_NID=agentNID, updated_date=CurrentDT
  WHERE agency_ID = agencyID:

GET DIAGNOSTICS rcount = ROW_COUNT;
  IF (rcount <> 1) THEN
    GOTO UPD_FAILURE;
  ELSE
    GOTO SUCCESS;
  END IF;
END IF;

SUCCESS:     RETURN 0;
INS_FAILURE:  RETURN 900;
UPD_FAILURE:  RETURN 901;
END;
DEPLOYMENT
moving procedures into production

- **remember: db2 implements sql procedures using c programs**
  - qsys2/syprocs catalog updated each time that an sql procedure is created
  - program object location found in external_name column

- **approach: save & restore program object that gets created by db2**
  - c program object is tagged so that db2 can recognize it as an sql stored procedure (in most cases - see next chart)
  - procedure created in restore library regardless of the create procedure source
  - if specific specified on create procedure statement, then it must be unique - system only generates unique specific name when it's not specified
  - program restore always completes, the database catalogs might not get updated
Moving Procedures into Production

• DB2 tries to automatically recognize the C program on the restore as an SQL Stored Procedure, but there are exceptions....
  – If DB2 does not find a matching procedure in the catalogs, then the C program is registered as an SQL Stored Procedure
  – If DB2 finds one procedure with the same name (differences in parameters ignored), catalog entries for the existing procedure are dropped and the new program object is registered as an SQL Stored Procedure.
  – If DB2 finds one or more procedures with the same name and a different signature (ie, different parms), then the restored program will be registered as a procedure with the same name \((\text{and possibly overlay the program object for the existing procedure})\)
• When parameters have changed it is probably best to drop the existing procedure before the restore
RESULT SETS
Result Sets & Procedures

- Stored procedures in combination with result sets can drastically reduce network trips by returning blocks of results.

- Stored procedures that return result sets can be called from any interface as of IBM i 7.1
  - Pre 7.1 releases could not use embedded SQL to process stored procedure result set.

- Result sets are returned via open SQL cursors
  - External stored procedures can also use arrays.
### Result Set Example

CALL RegionCustList(16)

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>FIRSTNME</th>
<th>LASTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>000050</td>
<td>JOHN</td>
<td>Geyer</td>
</tr>
<tr>
<td>000060</td>
<td>Irving</td>
<td>Stern</td>
</tr>
<tr>
<td>000070</td>
<td>Eva</td>
<td>Pulaski</td>
</tr>
<tr>
<td>000090</td>
<td>Eileen</td>
<td>Henderson</td>
</tr>
<tr>
<td>000130</td>
<td>Delores</td>
<td>Quintana</td>
</tr>
<tr>
<td>000150</td>
<td>Bruce</td>
<td>Adamson</td>
</tr>
<tr>
<td>000170</td>
<td>Masatoshi</td>
<td>Yoshimura</td>
</tr>
<tr>
<td>000190</td>
<td>James</td>
<td>Walker</td>
</tr>
<tr>
<td>000200</td>
<td>David</td>
<td>Brown</td>
</tr>
<tr>
<td>000260</td>
<td>Sybil</td>
<td>Johnson</td>
</tr>
<tr>
<td>000320</td>
<td>Randy</td>
<td>Brown</td>
</tr>
<tr>
<td>000340</td>
<td>Jason</td>
<td>Goetz</td>
</tr>
<tr>
<td>200170</td>
<td>Kim</td>
<td>Griffith</td>
</tr>
<tr>
<td>200340</td>
<td>Roy</td>
<td>Alonzo</td>
</tr>
</tbody>
</table>
CREATE PROCEDURE RegionCustList ( IN Region# INTEGER )
  RESULT SET 1
  LANGUAGE SQL

BEGIN
--Take the inputted region number, Region# and
--return the set of customers from that region
--via a result set

  DECLARE c1 CURSOR WITH RETURN TO CALLER FOR
      SELECT custnum, firstname, lastname
      FROM custtable WHERE region = Region#;

  OPEN c1;

END;
CREATE PROCEDURE RegionCustList ( IN Region# INTEGER )
    RESULT SET 1
    LANGUAGE SQL

BEGIN
--Take the inputted region number, Region# and
--return the set of customers from that region
--via a result set

    DECLARE c1 CURSOR FOR
        SELECT custnum, firstname, lastname
        FROM custtable WHERE region = Region#;

    OPEN c1;

    SET RESULT SETS CURSOR c1;
END;
Returning Result Sets

• After specifying a non-zero value on the RESULT SET clause there are several ways for a procedure to return result sets. (SQL Development Kit required)
  – SET RESULT SETS statement (non-standard) used to identify result sets - cursor or array
    • SET RESULT SETS CURSOR x2;
    • SET RESULT SETS ARRAY :tperf FOR :rowcntr ROWS (external only)
  – If SET RESULT SETS statement not specified
    • If no cursors use WITH RETURN clause, then any cursor still open at the end of the procedure is identified as a result set
    • If any cursors have specified WITH RETURN, then any cursor specifying WITH RETURN that's left open at the end of an SQL procedure is identified as a result set.

    DECLARE c2 CURSOR WITH RETURN TO CALLER FOR SELECT * FROM SYSTABLES;

• If multiple result sets, then result sets returned in the ordered specified on SET RESULT SETS or in the order that the cursors are opened.
Considerations

- **Result Set Consumer Control**
  - RETURN TO CLIENT
    DECLARE c1 CURSOR WITH RETURN TO CLIENT FOR SELECT * FROM t1
    SET RESULT SETS WITH RETURN TO CLIENT FOR CURSOR x1
  
  - RETURN TO CALLER
    DECLARE c1 CURSOR WITH RETURN TO CALLER FOR SELECT * FROM t1
    SET RESULT SETS WITH RETURN TO CALLER FOR ARRAY :array1 FOR :hv1 ROWS
Considerations

• Result Set Considerations:
  – If result set returned via cursor, rows are returned starting with the current position of the cursor
  – Typical call processing (ODBC)
    • Execute the stored procedure, then use SQLBindCol and SQLFetch against CALL statement handle
    • If multiple result sets then use SQLMoreResults to move to the next the result set
  – System i Navigator SQL Script Center best tool for debugging result sets and output parameter values
strcpy(stmttxt, "CALL Proc1(?,?,?)");
rc = SQLPrepare(hstmt, stmttxt, SQL_NTSM);

/* Ready procedure parameters */
rc = SQLBindParameter( hstmt, 1, SQL_PARAM_INPUT, SQL_INTEGER, SQL_INTEGER,
sizeof(Nmpd_Year), 0, (SQLPOINTER *)&Nmpd_Year,
sizeof(Nmpd_Year), (SQLINTEGER *) &Nmi_PcbValue );

/* call the procedure */
rc = SQLExecute(hstmt);

/* Bind columns for the results */
rc = SQLBindCol(hnd_Hstmt, 1, SQL_CHAR, (SQLPOINTER) Chr_Supplier_Name,
sizeof(Chr_Supplier_Name), (SQLINTEGER *) &Nmi_PcbValue );

/* Scroll thru the results */
while ( rc == SQL_SUCCESS )
{
    rc = SQLFetch(hstmt);
    if ( rc == SQL_SUCCESS )
    { /* Print current results */
        ...
    }
}

/* check for more result sets */
rc = SQLMoreResults(hstmt);
if ( rc <> SQL_NO_DATA_FOUND )
{ /* process the next result set */
    ...
}
IBM i 7.1 allows direct access of stored procedure result sets with Embedded SQL & SQL Routines

- Key Enabler Statements: ASSOCIATE LOCATOR & ALLOCATE CURSOR
- Optionally, DESCRIBE PROCEDURE & DESCRIBE CURSOR statements can be used to dynamically determine the number and contents of a result set

```sql
DECLARE sprs1 RESULT_SET_LOCATOR VARYING;
CALL GetProj(projdept);
ASSOCIATE LOCATORS(sprs1) WITH PROCEDURE GetProj;
ALLOCATE mycur CURSOR FOR RESULT SET sprs1;
myloop: LOOP
  FETCH mycur INTO prname, prstaff;
  IF row_not_found=1 THEN
    LEAVE fetch_loop;
  END IF;
  SET totstaff= totstaff + prstaff;
  IF prstaff > moststaff THEN
    SET bigproj = prname;
    SET moststaff= prstaff;
  END IF;
END LOOP;
CLOSE mycur;
```

<table>
<thead>
<tr>
<th>PROJNAME</th>
<th>PRSTAFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERAL ADMIN</td>
<td>6.00</td>
</tr>
<tr>
<td>PAYROLL PROG</td>
<td>2.00</td>
</tr>
<tr>
<td>PERSONNEL PROG</td>
<td>1.00</td>
</tr>
<tr>
<td>ACCOUNT PROG</td>
<td>2.00</td>
</tr>
</tbody>
</table>
TOOLS & DEBUG
Tool Considerations

- **System i Navigator**
  - A simple graphical editor for developing SQL Procedures
  - Runtime debug by displaying contents of result sets and output parameters
  - Integration with no charge IBM i Graphical Debugger
    - GUI version of STRDBG
    - Can be used with any IBM i program

- **IBM Data Studio**
  - More advanced graphical editor, also includes graphical debugger
  - Download trial at: ibm.com/software/data/db2/express/download.html
  - Connectivity requirements
    - Debugger requires usage of IBM Data Server JDBC Driver which requires...
    - DB2 Connect license
System i Navigator – Debugger Integration

>>>>>> Graphical Debugger White Paper:
ibm.com/servers/enable/site/education/abstracts/sqldebug_abs.html
SQL Procedure Debug

1 CREATE PROCEDURE KMTEST 

2    PP2 ( ) LANGUAGE SQL SET OPTION DBGVIEW

3 BEGIN

4 DECLARE X INT;

5 DECLARE Y INT;

6 SET X = 0;

7 SET Y = ABSVAL ( ( X + 1 ) * Y );

8 END;

17 SQLP_L1.Y = -9;

18 SQLP_L1.SQLP_I2 = 0;

19 if (SQLPROCH(&sqlca) == 1) {

20 memcpy(&sqlca_sav, &sqlca, sizeof(sqlca_sav));

21 memcpy(&sqlca, &sqlca_sav, sizeof(sqlca_sav));

22 goto SQL_END_PP3; }

154 ***

152 EXEC SQL SET :SQLP_L1.Y :SQLP_L1.SQLP_I2 = ABSVAL ( ( 

153 LP_L1.Y :SQLP_L1.SQLP_I2 ) 

154 ***
IBM i Debuggers – Enablement Steps

- **Use SET OPTION clause to enable SQL source-level debug**
  
  ```sql
  SET OPTION DBGVIEW = *SOURCE
  ```

- **Debuggers only support 10 character program names**
  - Use SPECIFIC clause to provide meaningful short name

  ```sql
  CREATE PROCEDURE LONG_PROCEDURE_NAME ( )
  SPECIFIC LONGPROC ...
  ```

- **Specify BEGIN label to enable EVAL command for local variables**

  ```sql
  CREATE PROCEDURE proc1(IN p1 INT)
  LANGUAGE SQL
  SET OPTION DBGVIEW=*SOURCE
  sp: BEGIN
  DECLARE x INT;
  SET x = p1 + 5;
  END;
  ```
IBM i Debuggers – EVAL commands

- Accessing SQL Routine variables & parameters

Parameters:
  EVAL P22.PARM1

Variables:
  EVAL SP
  EVAL SP.X
  EVAL *SP.Z :S 5

CREATE PROCEDURE p22(IN parm1 INT)
  LANGUAGE SQL
  SET OPTION DBGVIEW=*SOURCE
sp: BEGIN
  DECLARE x,y INT;
  DECLARE z CHAR(5);
  SET x = parm1;
  SET y = -9;
  SET y = ABSVAL((x+1)*y);
  SET z = 'ABCDE';
END;
PERFORMANCE
Procedural SQL—Performance Considerations

- Procedures and functions most effective from a performance perspective when multiple operations performed on a single invocation

- Minimize nested-SQL call statements to other routines
  - SQL Call implemented as unbound call

- No support for Blocked Fetches or Inserts
  - External procedures and functions may be a consideration

- Recreate SQL procedural objects on new releases to take advantage of DB2 enhancements for C code generation
  - Major improvements made in V5R4, some enhancements in 6.1
Procedural SQL—Performance Considerations

• Utilize coding practices that improve efficiency of the C code generated by DB2. Best practice resources:
  – Improving SQL Procedure Performance white paper (ibm.com/systems/i/db2/awp.html)

  – Local variable suggestions
    • Declare local variables as not null
    • Use integer instead of decimal precision with 0
    • Minimize the usage of character & date variables
    • Use the same data type, length and scale for numeric variables that are used together in assignments

  – Consider moving handlers for a specific condition/statement within a nested compound statement

```sql
BEGIN
  DECLARE CONTINUE HANDLER
  FOR SQLSTATE ' 23504'

  ... DELETE FROM master WHERE id=1;
  ...
END
```

```sql
BEGIN
  ...
  BEGIN
    DECLARE CONTINUE HANDLER FOR SQLSTATE ' 23504'

    DELETE FROM master WHERE id=1;
  END

  ...
END
```
Additional Information

- **DB2 for i Websites**
  - Home Page: [ibm.com/systems/i/db2](ibm.com/systems/i/db2)
  - DeveloperWorks Zone: [ibm.com/developerworks/db2/products/db2i5OS](ibm.com/developerworks/db2/products/db2i5OS)
  - Porting Zone: [ibm.com/servers/enable/site/db2/porting.html](ibm.com/servers/enable/site/db2/porting.html)

- **Newsgroups**
  - USENET: comp.sys.ibm.as400.misc, comp.databases.ibm-db2

- **Education Resources - Classroom & Online**
  - [ibm.com/systemi/db2/gettingstarted.html](ibm.com/systemi/db2/gettingstarted.html)
  - [ibm.com/partnerworld/wps/training/i5os/courses](ibm.com/partnerworld/wps/training/i5os/courses)

- **DB2 for i Publications**
  - White Papers: [ibm.com/partnerworld/wps/whitepaper/i5os](ibm.com/partnerworld/wps/whitepaper/i5os)
  - Online Manuals: [ibm.com/systemi/db2/books.html](ibm.com/systemi/db2/books.html)
  - DB2 for i Redbooks ([ibm.com/systems/i/db2/relredbooks.html](ibm.com/systems/i/db2/relredbooks.html))
    - *Stored Procedures, Triggers, and UDFs on DB2 UDB for iSeries* (SG24-6503)
    - *OnDemand SQL Performance Analysis ... in V5R4* (SG24-7326)
EXTERNAL STORED PROCEDURES
External Procedures & Programs

- External Procedure- high-level language program approach
  
  ```sql
  CREATE PROCEDURE PROC1 (CHAR(4), INT)
  EXTERNAL NAME TESTLIB/SPPGM
  LANGUAGE RPG GENERAL
  ```

- DB2 directed to use RPG program, SPPGM, as a Stored Procedure when Stored Procedure, PROC1, is invoked.
  - RPG Program may or may not have contained embedded SQL
  - ILE program objects will be tagged as stored procedures

- **External program approach gives great flexibility**
  - Can perform native IO in external procedure
  - Can take advantage of existing RPG, COBOL, etc skills to write stored procedures

- **Supported languages**
  - C, C++, RPG, COBOL, Java, CL

- Service programs objects also supported
**External Procedure Parameter Styles**

- **SQL & DB2SQL** essentially the same - DB2SQL offers additional system parameter (DBINFO)

- **GENERAL/GENERAL WITH NULLS** essentially the same except for null value indicator array passed on "WITH NULLS"

- **DB2GENERAL & JAVA** only available for external Java Stored Procedures
Parameter Styles

- **With GENERAL style**, parameters just match the IN, INOUT, & OUT parameters specified on the CREATE PROCEDURE variables
  - GENERAL WITH NULLS adds N parameters for indicator variables
- **Default style is SQL** - parameter style pretty close to GENERAL WITH NULLS
  - First N parameters match those specified on the procedure definition
  - N parameters for indicator variables
  - CHAR(5) for SQLSTATE
    - External program can assign a value
    - (00000-Success, 01HXX-Warning, 38XXX-Error)
  - VARCHAR(517) for fully qualified procedure name
  - VARCHAR(128) for specific name
  - VARCHAR(70) output variable for message text
    - Message text can only be set when SQLSTATE is 38XXX
Parameter Styles

- **DB2SQL** superset of the SQL parameter style when the DBINFO keyword is specified. DBINFO only can be specified with the DB2SQL style.

- DBINFO tells DB2 to pass an additional parameter to the stored procedure in a structure containing the following database information:
  - Relational database name
  - Authorization ID
  - CCSID
  - Version & Release
  - Platform
CREATE PROCEDURE sptest.stylesql
  (IN inparm char(6), OUT outparm char(6))
  LANGUAGE RPGLE EXTERNAL NAME sptest.mypgm
  PARAMETER STYLE SQL;

...  // Extra SQL style parms NOT passed on the CALL statement
exec sql
  CALL sptest.stylesql(:inpVar :inpIndicator, :outVar :outIndicated);
...

  // Hidden parameters accessed by caller via SQLCA on return
if SQLCODE = -443 and SQLSTATE= '38999';
  { ... do error handling ... }

  // User-defined message would be accessed by finding the
  // sixth token in sqlca.sqlerrmc
  { ... process returned message text ... }
...
## Procedure source - SQL Parameter Style

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
<th>Length</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>D inParm</td>
<td>For parm - input</td>
<td>s 6a</td>
<td></td>
</tr>
<tr>
<td>D outParm</td>
<td>For parm - output</td>
<td>s 6a</td>
<td></td>
</tr>
<tr>
<td>D inParmInd</td>
<td>Ind parm - input</td>
<td>s 5i 0</td>
<td></td>
</tr>
<tr>
<td>D outParmInd</td>
<td>Ind Parm - output</td>
<td>s 5i 0</td>
<td></td>
</tr>
<tr>
<td>D sqlState</td>
<td>SQL State Parm</td>
<td>s 5a</td>
<td></td>
</tr>
<tr>
<td>D procName</td>
<td>Proc name parm</td>
<td>S 517a varying</td>
<td></td>
</tr>
<tr>
<td>D specName</td>
<td>Specific name parm</td>
<td>s 128a varying</td>
<td></td>
</tr>
<tr>
<td>D sqlMsgTxt</td>
<td>Message text parm</td>
<td>s 70a varying</td>
<td></td>
</tr>
</tbody>
</table>

```c
C *entry
C parm inParm
C parm outParm
C parm inParmInd
C parm outParmInd
C parm sqlState
C parm procName
C parm specName
C parm sqlMsgTxt
```

...(CONTINUED)...
// Check the procedure name parameter
if %len(procname) = 15 and
   procname = 'SPTEST.STYLESQL';
   // ...passed in the correct procedure name...
endif;

// Check the Input Parameter Value
if inParm = 'INPVAL';
   // Pass back input value as the output parameter value
   outParm = inParm;
endif;

// Else set SqlState and message output parameters
// Set message text variable - only can be set when
// state parm set to 38xxx
else;
   sqlState = '38999';
   sqlMsgTxt = 'Possible problem encountered';
endif;

   ...
return;
/end-free
External Procedures - Error Handling

• **How are errors handled in external procedures (w/GENERAL) ?**
  - Procedures using GENERAL style cannot directly pass back error information
  - Invoker will always receive SQLCODE=0 & SQLSTATE='00000'
    • Error feedback is in the form of joblog messages

• **How are errors handled in external procedures (w/SQL) ?**
  - If a procedure doesn't trap or handle the error/warning, then invoker returned SQLCODE=0 & SQLSTATE='00000'
    • Error feedback is in the form of joblog messages
  - If procedure does have logic to trap or handle the error/warning then it is possible to notify invoker of error/warning event (SQLCODE will equal -443)
    • For errors, procedure can set SQLSTATE parm (and optional message text parm) to user-defined SQLSTATE value in the format of 38xxx
    • For warnings, procedure can set SQLSTATE parm to user-defined SQLSTATE value in the format of 01Hxx
    • Setting the SQLSTATE parameter to any other value (other than 01Hxx or 38xxx), even system SQLSTATEs, will cause DB2 to return an SQLSTATE value of '39001'

• **NOTE:** Output parameter values are NOT guaranteed to be returned to the invoker once an error is encountered.