SQL Routines – Procedures, Functions (and Triggers)

Rob Bestgen
bestgen@us.ibm.com
IBM - DB2 for i

SQL as a development language

SQL
- Is a well established, standardized language for database access

However, there is more than just database query access
- SQL PSM provides a full procedural programming language in its own right
  - Enhances portability
    - Supported across DB2 Family – DB2 implementation follows the SQL PSM Standard
    - Similar to proprietary procedure languages available from other DBMS (PL/SQL, T-SQL, etc)
    - First introduced way back at V4R2

  - Makes it easier for SQL programmers to be productive faster on IBM i
What’s new

There is a refreshed procedures RedBook available!

http://www.redbooks.ibm.com/redpieces/abstracts/sq248326.html

So new it’s still in draft!

Up to date content covers the latest through v7.2

SQL Routines

- SQL Routines
  - Procedures
  - Functions

- SQL Triggers included in this discussion

- Created using the SQL programming language

- Enhanced portability
  - Supported across DB2 Family
  - Follows SQL PSM Standard
  - Similar functionality to proprietary languages on other DBMS (PL/SQL, T-SQL, etc)

- Makes it easier for SQL programmers to be productive faster on IBM i
DB2 for i supports two types of procedures/functions/triggers

1. SQL
   • Coded entirely with SQL following (PSM) Standard
   • Allows full suite of SQL statements

2. External
   • Register a high-level language program (RPG, COBOL, Java, C…) for subsequent calls from SQL**
   • may or may not use SQL within the program itself

** Non-SQL triggers use CL command ADDPFTRG
Introduction to SQL Routines

- Comparison of SQL Routine types
  - Stored Procedures
    - Similar to high-level language program, facilitate reuse of common business logic or business process
    - Most commonly used to call procedure on another system
      - Minimize network traffic
      - Secure data access
    - Invoked with SQL CALL statement by developer
  - Triggers
    - Mechanism to guarantee that same business process occurs on every data change operation – regardless of interface
    - Invoked by DB2
  - User-Defined Function
    - Most commonly called for each row being processed during SQL statement execution
    - Invoked from SQL statement by developer
      SELECT myfunc1() FROM mytable WHERE myfunc2(col1)>100

What is a Stored Procedure?

- Like a called program, except
  - Called from SQL-based interfaces via the SQL CALL statement
  - Traditional input and output parameters along with result set capability

- Follows IBM i security model for programs
  - IBM i adopted authority model can be leveraged

- CREATE PROCEDURE SQL statement to create

- CALL SQL statement to invoke

- A unique feature of procedures compared to programs is that they can return result sets
  - A virtual table answer set
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 over and over

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

What is a Stored Procedure?

- Provides performance savings in distributed computing environments
  - One request initiates multiple transactions and processes

- Performance improvements further enhanced by the option of providing result sets back to
  - ODBC, JDBC, .NET & CLI clients
  - High level languages: RPG, COBOL, C
What is a Function?

Two types:

1. Scalar function
   - Returns a single value
   - Invoked from almost any SQL statement
   - Good for encapsulating complex operations or calculations

2. Table function
   - Returns a set of rows with columns
     • A dynamic table!
   - Invoked from an SQL query or view (FROM clause)
   - Good for encapsulating non-traditional data or complex operations

- CREATE FUNCTION SQL statement to create
- Reference directly or wrap in TABLE statement to invoke*

* TABLE statement for table functions

Recipe for an SQL Scalar Function

1  Create it (one time)

   CREATE FUNCTION  Discount(totalSales DECIMAL(11,2))
   RETURNS DECIMAL(11,2)
   LANGUAGE SQL DETERMINISTIC
   BEGIN
     IF totalSales>10000 THEN
       RETURN totalSales*.9;
     ELSE
       RETURN totalSales;
     END IF;
   END

2  Call it from an SQL query or statement

   SELECT SUM(sales), Discount( SUM(sales) ) FROM mysales…
Recipe for an SQL Table Function

1. Create it (one time)

   CREATE FUNCTION bestSales (BonusThreshold DECIMAL(11,2))
     RETURNS TABLE (Empid INT,
                     LastName VARCHAR(30),
                     FirstName VARCHAR(30))
   LANGUAGE SQL
   READS SQL DATA
   RETURN
   SELECT id, lname, fname FROM custsales
   GROUP BY id, lname, fname
   HAVING SUM(sales)>BonusThreshold

2. Call it from an SQL query or statement

   SELECT LastName, Empid from TABLE(bestSales(100000)) T1

What is an SQL Trigger?

A program called when row(s) in a table or file change
   - Associated with a table (or view)
   - Invoked automatically by DB2 when a row changes

When do you need triggers?
   - Consistently enforce business rules
   - Monitor critical tables

CREATE TRIGGER SQL statement to create and register*

* ADDPFTRG CL command for external triggers
Trigger events

- Row changes
  - Insert
  - Update
  - Delete
  - Multiple statement

- SQL Triggers support additional controls
  - Column-level
  - Statement-level

- Timing
  - Before
    - Before Triggers can change application data before record modification
  - After
  - Instead of SQL trigger – view only
    - Directs insert/update/delete to the trigger, no actual change to underlying table

SQL Trigger Examples

```
CREATE TRIGGER protect_salary
BEFORE UPDATE
OF salary ON employee
REFERENCING NEW AS n OLD AS o
FOR EACH ROW
WHEN (n.salary > 1.5 * o.salary)
SET n.salary = 1.5 * o.salary;
```

```
CREATE TRIGGER big_spenders
AFTER INSERT ON expenses
REFERENCING NEW ROW AS n
FOR EACH ROW
BEGIN
DECLARE emplname CHAR(30);
IF (n.totalamount > 10000) THEN
  SET emplname = (SELECT lname FROM emp WHERE empid = n.empno);
  INSERT INTO travel_audit
  VALUES(n.empno, emplname, n.deptno, n.totalamount, n.enddate);
END IF;
END
```
Multi-action SQL Triggers

- A single SQL Trigger can be created to handle multiple database operations
  - Enables centralization of all trigger logic
  - Can be used to simplify management and installation of triggers

```sql
CREATE TRIGGER employee_monitor
  AFTER INSERT OR DELETE OR UPDATE OF salary
  ON employee
  REFERENCING NEW AS n OLD AS o
  FOR EACH ROW
  BEGIN
    IF INSERTING THEN
      UPDATE company_stats SET nbremp=nbremp + 1;
      END IF;
    IF DELETING THEN
      UPDATE company_stats SET nbremp=nbremp - 1;
      END IF;
    IF UPDATING AND (n.salary > 1.1 * o.salary)
      THEN SIGNAL SQLSTATE '75000'
            SET MESSAGE_TEXT = 'Salary increase > 10%';
      END IF;
  END
```

A Common Language
Create Options

CREATE OR REPLACE ... routine name ...

- LANGUAGE - Language of procedure (SQL, C, RPG...)
- SPECIFIC - unique "short" name for SQL procedures & UDFs **
  - Can be used to control underlying *PGM object name
- FENCED/NOT FENCED – Only valid for Functions & Procedures
  - Use NOT FENCED when possible for better performance

** PROGRAM NAME clause available for SQL Triggers

Create 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)

Note: Create routine at the ‘lowest’ option possible for your situation
  - Lowest to highest: NO, CONTAINS, READS, MODIFIES

* Option not applicable to Triggers
Create Options

- **DETERMINISTIC**
  - procedure/function will always return the same result from successive calls with identical input arguments
  - Best performing option, but make sure it is true!

- **PROGRAM TYPE**
  - **SUB**: creates a service program object (better performance)
  - **MAIN**: creates a program object

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 Routines are:
  - **USRPRF** for adopted authority (defaults to *OWNER)
  - **DBGVIEW** for creating debuggable version of SQL Procedure
    - *SOURCE enables SQL statement-level debug
SQL Compound Statement

- **ATOMIC**
  - all statements succeed or are rolled back.
  - COMMIT or ROLLBACK cannot be specified in the routine
  - must also be created with COMMIT ON RETURN YES
- **NOT ATOMIC** – no guarantee or atomicity

### Declarations
- **<declare variables>**
- **<declare conditions>**
- **<declare cursors>**
- **<declare handlers>**

### Logic - Can contain nested compound statements

- **BEGIN**
- **END**

### Basic Constructs

- **DECLARE** – define variable. Initialized when procedure is called
  ```sql
  DECLARE v_midinit, v_edlevel CHAR(1);
  DECLARE v_ordQuantity INT DEFAULT 0;
  DECLARE v_enddate DATE DEFAULT NULL;
  ```
  - Uninitialized variables set to NULL
  - Single-dimension arrays
    ```sql
    CREATE TYPE pids AS CHAR(3) ARRAY[];
    CREATE TYPE intarray AS INTEGER ARRAY[5];
    ```

- **SET** - assigning a value parameter or variable
  ```sql
  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, rest of line is a comment
  - `/*...*/` Bracketed comments, within brackets is a comment
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 query

  Ex:
  
  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 SELECT statement to be accessed directly!
  - Cursor can be used in WHERE CURRENT OF... operation
Looping Constructs

- LOOP, REPEAT and WHILE statements

**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 and Error Handling

Routines provide a rich set of error and message handling capabilities

- GET DIAGNOSTICS
- SQLSTATE and SQLCODE variables
- CONDITIONS and HANDLERs
- SIGNAL and RESIGNAL
- RETURN statement
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:
      ```sql
      DECLARE update_counter INTEGER;
      ...
      UPDATE orders SET status='LATE'
          WHERE ship_date < CURRENT DATE;
      GET DIAGNOSTICS update_counter = ROW_COUNT;
      ...
      ```

- **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 'xxyyzz'
    - predefined: SQLWARNING, NOT FOUND, SQLEXCEPTION
Feedback & Error Handling Example

CREATE PROCEDURE proc1()
...
BEGIN
DECLARE at_end CHAR(1) DEFAULT 'N';
-- 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 < '01/01/1990';
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 info
      - 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 optional
    - Condition info – condition name or SQLSTATE ‘xxyzz’
    - SET clause provides ability to pass along additional diagnostic info

- **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)
**Signal & Resignal Example**

```sql
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**

- RETURN statement can be used to communicate high-level success/failure status to caller
  - RETURN <optional integer value>;
  - If no return statement not specified, then...
    - If SQLCODE >= 0, then return value set to a value of 0
    - If SQLCODE < 0, then return value set to -1
- Accessing the return value
  - when invoked by another procedure
    ```sql
    GET DIAGNOSTICS  statusvar = RETURN_STATUS;
    ```
  - "?=CALL <procedure name>" syntax common in ODBC and JDBC
  - Returned in SQLERRD[0]

```sql
CREATE PROCEDURE ModAgency(IN agencyVID INTEGER,
IN agencyNUM  INTEGER, IN agencyID INTEGER, IN agentNID INTEGER)
... BEGIN ...
SUCCESS:  RETURN 0;
INS_FAILURE: RETURN 900;
UPD_FAILURE: RETURN 901;
END;
```
Mixing Static and Dynamic SQL

It is possible, and quite common, to mix Static SQL and Dynamic SQL in the same procedure (or Function):

- **Static** – Things you know about during the procedure creation

- **Dynamic** – to handle things that can vary

Static and Dynamic mix example

```sql
CREATE OR REPLACE PROCEDURE Process_Table
    (DATALIB VARCHAR(128), DATATABLE VARCHAR(128))
LANGUAGE SQL
MODIFIES SQL DATA
SET OPTION COMMIT = "NC"
BEGIN
    DECLARE NF INT DEFAULT 0;
    DECLARE EOF INT DEFAULT 0;
    DECLARE D_SQL VARCHAR(3000);
    DECLARE D_ITEM_KEY CHAR(8);
    DECLARE NOTFOUND CONDITION FOR '42704';
    DECLARE CONTINUE HANDLER
        FOR NOTFOUND SET NF = 1;
    DECLARE CONTINUE HANDLER FOR
        SQLSTATE '02000' SET EOF = 1;
    SET D_SQL = 'SELECT ITEM_KEY FROM ' CONCAT DATALIB CONCAT '.' CONCAT DATATABLE;
    PREPARE ITEM_P FROM D_SQL;
    BEGIN
        DECLARE ITEM_CURSOR CURSOR FOR
            ITEM_P;
        OPEN ITEM_CURSOR;
        IF NF=1 THEN
            RETURN;
        END IF;
        FETCH ITEM_CURSOR INTO D_ITEM_KEY;
        FETCHLOOP: WHILE EOF=0
            DO
                ...
            END FETCHLOOP;
        CLOSE ITEM_CURSOR;
    END;
END;
```
Unique Features

Create Procedure options

- Named and default parameters support for SQL procedures
  - Extend procedures without breaking existing callers
  - Simplify calling requirements by adding default values
  - Procedure call invocation has the same usability as CL commands

```
default-clause:
  DEFAULT
  constant
  special-register
  global-variable
  (expression)
```

Examples:

**Ex1:**
CREATE PROCEDURE p1 (i1 INT, i2 INT DEFAULT 0, i3 INT DEFAULT -1)...
CALL p1(55)
CALL p1(55, i3=>33)

**Ex2:**
CREATE OR REPLACE PROCEDURE Add_New_Employee
(EmpFirstName VARCHAR(20), EmpLastName VARCHAR(20), EmpAddress VARCHAR(200),
EmpTwitterHandle VARCHAR(40) DEFAULT NULL,
EmpUniqueID BIGINT DEFAULT (SELECT NEXT VALUE from EmployeeIDs))
Create Procedure options

CREATE OR REPLACE PROCEDURE procedure name
  (IN parm name data type, 
   OUT parm name data type, 
   INOUT parm name data type)

- Procedure name + number of parameters make a unique signature
  - Can have multiple procedures with the same name within a library
    • NOTE: SPECIFIC names must be unique

Note: default parameters support in v7r1+ and V7R2 makes this less heavy handed

RESULT SETS specifies max number of result sets that can be returned from procedure.
- Returned to ODBC, JDBC, .NET & CLI clients
- Result sets returned to Embedded SQL in other languages e.g. RPG
Create Procedure options - V7R2

- Enables a stored procedure’s database changes to be executed independently of the calling application
  - DB2 automatically issues Commit/Rollback for autonomous procedure

```
CREATE PROCEDURE update_agency(IN agencyVID INTEGER, IN agencyNUM INTEGER, IN agencyID INTEGER, IN agentNID)
LANGUAGE SQL
AUTONOMOUS
BEGIN
  UPDATE agency
  SET agency_vid=agencyVID, agency_num=agencyNUM, agent_NID=agentNID, updated_date=CURRENT_TIMESTAMP
  WHERE agency_ID = agencyID;

  INSERT INTO agency_log
  VALUES(USER, CURRENT_TIMESTAMP, agencyVID, agencyNUM, agentNID);
END
```

Create Function options

```
CREATE OR REPLACE FUNCTION function name ()
  parm name data type
BEGIN
  -- Function body
END;
```

- Function name + number of parameters + parameter (family) data type make a unique signature
  - Enables overloading / polymorphism for functions within the same library
    - Ex: func1(parm1 int) & func1(parm1 char)

Note: casting rules have changed (softened) in v7r2
Create Function V7R2 - parameter defaults

- Default parameters supported for functions in v7r2
  - Parameters can be omitted from invocation when default value defined
  - Parameters may be specified in any order by specifying the parameter name in the call
  - Catches up with Procedures (added in V7R1)

```
CREATE FUNCTION New_Prescription (  
  drugName CHAR(40),  
  prescID INTEGER DEFAULT (VALUES(NEXT VALUE FOR idSequence)),  
  refills INTEGER DEFAULT 0) ...
```

Omitting parameters – defaults
Using a named parameter

```
New_Prescription('Accutane')
```

```
New_Prescription('Accutane', Refills=>3)
```

Create Trigger Options

- Trigger Granularity
  - FOR EACH ROW
    - Invoked for each row satisfying trigger condition
  - FOR EACH STATEMENT
    - Invoked once per statement, regardless of the number of rows processed
    - Not valid with Before triggers or Trigger Mode DB2ROW
  - Column-level control
    - for UPDATE trigger event
    - Only an update referencing the listed columns causes trigger to fire
Create Trigger Options …

- Transition Tables
  - Temporary table that contains the before and/or after images of all affected rows
    - OLD TABLE - Before image of all affected rows
    - NEW TABLE - After image of all affected rows
  - Example:
    ```sql
    REFERENCING OLD TABLE AS oldtbl
    ...(SELECT COUNT(*) FROM oldtbl)...
    ```
  - A single SQL statement can process multiple rows
  - Use in FOR EACH STATEMENT triggers

- Transition Variables
  - Provides access to before and after image of record
    - OLD ROW - Before image of row (update and delete)
    - NEW ROW - After image of row (update and insert)
  - Example:
    ```sql
    ...REFERENCING OLD ROW AS oldrow REFERENCING NEW ROW AS newrow...
    ...IF newrow.salary > oldrow.salary + 10000...
    ```
  - Use in FOR EACH ROW triggers

 Stored Procedure
 Result Sets
Result Sets & Procedures

- Stored procedures can return answer sets using result sets
- Returned result sets can be consumed by:
  - System i Access ODBC, OLE DB & ADO.NET middleware
  - SQL CLI
  - Toolbox JDBC driver
  - Native JDBC driver
  - DB2 Connect
  - IBM i DRDA Connections
  - Embedded SQL & SQL Routines
  - RPG, COBOL, C!
- Result sets are returned via open SQL cursors

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;

** Proprietary statement SET RESULT SETS CURSOR xx also supported
Result Set Considerations

Result Set Consumer Control

- RETURN TO CLIENT

Ex: DECLARE c1 CURSOR WITH RETURN TO CLIENT FOR SELECT * FROM t1

- RETURN TO CALLER

Ex: DECLARE c1 CURSOR WITH RETURN TO CALLER FOR SELECT * FROM t1

Result Set Consumption: Embedded SQL & SQL Routines

- Directly retrieve result sets with embedded SQL & SQL Routines
  - ASSOCIATE LOCATOR & ALLOCATE CURSOR statements

```sql
DECLARE sprs1 RESULT_SET_LOCATOR VARYING;
CALL GetProjs(projdept);
ASSOCIATE LOCATOR (sprs1) WITH PROCEDURE GetProjs;
ALLOCATE mycur CURSOR FOR RESULT SET sprs1;
SET totstaff=0;
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;
```

Easy Integration!

** DESCRIBE PROCEDURE & DESCRIBE CURSOR statements can be used to dynamically determine the number and contents of a result set**
Programming considerations

Resolution of Procedure Calls

- Resolution of unqualified procedure and function invocations uses **SQL Path**
  - Default **path** values:
    - System Naming: *LIBL
    - SQL Naming: QSYS, QSYS2, SYSPROC, SYSIBMADM, authorization-ID
  - Changing default schema with SET CURRENT SCHEMA has NO impact on SQL Path
  - SQL Path hard-coded at **creation time** for Procedure & Function calls on Static SQL statements embedded within

- Procedures & Functions support overloading which further complicates resolution of unqualified invocations
  - Enables multiple versions of a procedure & function can have the same name within a schema
Protecting Source Code

- Source code for SQL procedural objects automatically stored in DB2 source code
  - Some algorithms considered intellectual asset
  - Customers can easily access intellectual asset:
    
    ```sql
    SELECT routine_definition FROM qsys2/sysroutines
    WHERE routine_name = 'MY_ROUTINE'
    ```

- Obfuscation can be used to mask source code for protection. Two methods available:
  - WRAP Function – generate obfuscated version of CREATE statement
  - CREATE_WRAPPED Procedure – creates obfuscated version of procedure/function

WRAP Obfuscation Function

- Returns obfuscated version of SQL CREATE statement that can be run on client’s system
  - Returns obfuscated statement as CLOB value
  - During product installation obfuscated statement executed to protect source code stored in catalog

```
VALUES( SYSIBMADM.WRAP
  ('CREATE PROCEDURE chgSalary(IN empno CHAR(6))
  LANGUAGE SQL BEGIN
   UPDATE employee SET empsal = empsal*(1 + .05*empjobtype)
   WHERE empid = empno;   END') );
```

```
CREATE PROCEDURE chgSalary (IN EMPNO CHAR(6)) WRAPPED
QSQ07010
aacxW8pW8FjG8pnG8VzG8FD68Fj68:Hi8:dY_pB2qpdW8pdW8pdW_praqe
baqebaGEMj_vsbPs5bOJUqHvayE1ogAlGWqz2jCIE1dQEjt33hd5Sps5
cYGViD1urv7vGKeOc4CwpCibb
```
CREATE_WRAPPED Obfuscation Procedure

- Creates obfuscated version of procedure/function* to protect source code stored in DB2 catalog
  - COMMIT level other than *NONE may be required when calling
    CREATE_WRAPPED procedures on some SQL interfaces

```
CALL SYSIBMADM.CREATE_WRAPPED(
  'CREATE PROCEDURE chgSalary(IN empno CHAR(6))
  LANGUAGE SQL
  BEGIN
    UPDATE employee SET empsal = empsal*(1 + .05*empjobtype)
    WHERE empid = empno;
  END');
```

* And triggers in v7.2

Obfuscation Considerations

- Obfuscated version of CREATE statement can be up to one-third longer than original SQL statement
- Obfuscation provides protection, but it’s not a strong form of encryption
- Obfuscated routines cannot be debugged
  - Clients would need to be provided with a debuggable version to use temporarily
- Obfuscation is preserved if associated program object is saved and restored onto another system
Additional Information

- **DB2 for i Websites**
  - Homepage: ibm.com/systems/power/software/i
  - Technology Updates
    www.ibm.com/developerworks/ibmi/techupdates/db2

- **Redbook**
  - www.redbooks.ibm.com/redpieces/abstracts/sg248326.html

- **Forums**
  - developerWorks:

- Articles on procedure resolution related to default parameters

Thank you!
Appendix
Performance Considerations

Multi-phase Creation Process

- Debug Considerations
  - Use DBGVIEW=*SOURCE to debug “original” SQL code
  - Remove from production version

- Call overhead
  - Program type SUB creates a SRVPGM, with improved call performance

- Performance Considerations
  - DB2 for i implements SQL procedural statements with a variety of techniques
  - Coding choices impact what method DB2 can use
Performance considerations

- SQL PL supports simple to complex constructs
  - Assignments, conditions, loops, complex data access

- DB2 can always run a statement as a query
  - SET A=1 → SELECT 1 INTO :A from QSQPTABL

- But DB2 for i implements SQL procedural statements with one of the following methods:
  - Base performance - Generated SELECT statement
  - Good performance - Expression Evaluator
  - Better performance - Inlining
  - Best performance - Pure C code generation

Pure C code generation

Generate pure C code
- For simple numeric operations that can be expressed in C
  - assignments: SET A=1;
  - Numeric operations SET A=B+1
INLINING

Pull statement from within a UDF into invoking statement
– Avoids call overhead
– Allows reordering of flow for better performance

Requirements
– IBM i 7.1 or later
– UDF has only a RETURN statement
  • DETERMINISTIC, NO EXTERNAL ACTION, NOT FENCED
– No table references
– No LOBs

```
CREATE FUNCTION myfunc(inval int) RETURNS int
    LANGUAGE SQL
    NOT FENCED DETERMINISTIC NO EXTERNAL
    ACTION
    RETURN inval*10;
...
SELECT myfunc(mycol) FROM mytbl
```

With inlining:

```
SELECT mycol*10 FROM mytbl
```

Expression Evaluator

- A fast path evaluator for 'table-less' SQL expressions in procedures
  - Scalar expressions in procedural statements (e.g. IF, SET)
  - Performs statement with a simple internal DB2 call

- Avoids overhead of generated SELECT
  - No need for Open/Fetch/Close processing
  - ODPs no longer required for simple expressions
    • Avoids potentially large number of reusable ODPs over QSQPTABL table

- Requirements
  - No table references
  - No LOBs
  - No UDFs (unless inlining occurs)
  - Must run with SQE (SQL Query Engine) Meaningful to IBM i 5.4
Tracking Performance overhead

- Use the database SQL performance monitor
  - See if rewritten SELECT statement is avoided
  - Fewer SQL statements run in general
  - Expressions implemented with C code are not collected in database monitor

- Can also look at job’s ODP list to see if it is reduced

SQL Procedure Performance Resources

- Improving SQL Procedure Performance White Paper

- System i Network article

- Routines chapter in IBM SQL Programming Guide
Trademarks and Disclaimers

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

IT Infrastructure Library is a registered trademark of the Central Computer and Telecommunications Agency which is now part of the Office of Government Commerce.

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S. Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Cell Broadband Engine and Cell/B.E. are trademarks of Sony Computer Entertainment, Inc., in the United States, other countries, or both and are used under license therefrom.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Information is provided "AS IS" without warranty of any kind.

The customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. Actual environmental costs and performance characteristics may vary by customer.

Information concerning non-IBM products was obtained from a supplier of these products, published announcement material, or other publicly available sources and does not constitute an endorsement of such products by IBM. Sources for non-IBM list prices and performance numbers are taken from publicly available information, including vendor announcements and vendor worldwide homepages. IBM has not tested these products and cannot confirm the accuracy of performance, capability, or any other claims related to non-IBM products. Questions on the capability of non-IBM products should be addressed to the supplier of those products.

All statements regarding IBM future directions and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

Some information addresses anticipated future capabilities. Such information is not intended as a definitive statement of a commitment to specific levels of performance, function or delivery schedules with respect to any future products. Such commitments are only made in IBM product announcements. The information is presented here to communicate IBM's current investment and development activities as a good faith effort to help with our customers' future planning.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Prices are suggested U.S. list prices and are subject to change without notice. Contact your IBM representative or Business Partner for the most current prices in your geography.

© 2016 IBM Corporation