SQL Procedures
(and Functions and Triggers)

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

SQL as a development language

SQL is a well established, standardized language for database access

SQL is also a programming language!

- **SQL/PSM** (https://en.wikipedia.org/wiki/SQL/PSM) is a full procedural programming language
- PSM enhances portability
  - Supported across DB2 Family
  - Similar to proprietary DBMS procedure languages (PL/SQL, T-SQL, etc…)

Makes it easier for SQL programmers to be productive faster on IBM i

Supported languages

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

IBM Power Systems

Supported languages

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

SQL is the main focus here

** Non-SQL triggers use CL command ADDPFTRG

SQL Routines

Comparison of SQL Routine types

- **Procedures**
  - Similar to high-level language program, facilitates reuse of common logic
  - Usage is similar to any program
  - Frequently used as the backend 'service' to application calls e.g. REST calls
    - Lessen network traffic
    - Secure data access
  - Can also return result sets of data
- **Functions** (user-defined)
  - Returns a result each time it is invoked
  - Frequently invoked from within an SQL query

    SELECT myfunc1() FROM mytable WHERE myfunc2(col1)>100
- **Triggers**
  - Attached to a file/table. Invoked for data changes
  - Automatically invoked by DB2 when necessary, regardless of interface

SQL Routines…

Each time an SQL Routine is created, DB2 for i uses the SQL procedural source to generate a C program (or srvpgm) object

- Developer does not need to know C code
- C compiler purchase is not required
- Like other pgms, IBM i security model and adopted authorities apply

```
CREATE PROCEDURE proc1 (IN Emp# CHAR(4),IN NwLvl INT)
LANGUAGE SQL
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
```

Procedure

1 Create it

```
CREATE OR REPLACE 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

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


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

Scalar Function

1 Create it (one time)

```
CREATE OR REPLACE 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…
```


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
```

Trigger

A program called when a 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

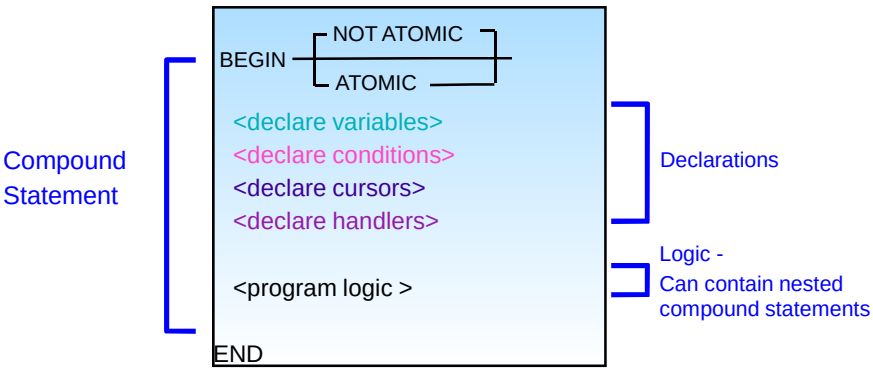
SQL Trigger Examples

```
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
```

A Common Language

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 of 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 set to NULL
  - Single-dimension arrays

```
CREATE TYPE pids AS CHAR(3) ARRAY[ ];
CREATE TYPE intarray AS INTEGER ARRAY[5];
```

- **SET** - assigning a variable or parameter

```
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!

Default Parameters

- Both Procedures and Functions (as of v7r2) support default parameters
  - Parameters can be omitted from invocation when a default value is defined
  - Parameters may also be specified in any order by specifying the parameter name in the call

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

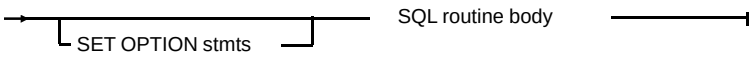
Omitting parameters – defaults

New_Prescription('Accutane')

Using a named parameter

New_Prescription('Accutane', **refills=>3**)

SET 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

Error Handling and Feedback

Feedback and Error Handling

Routines provide a rich set of error and message handling capabilites

- 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:

```
DECLARE update_counter INTEGER;
  ...
UPDATE orders SET status='LATE'
   WHERE ship_date < CURRENT DATE;
GET DIAGNOSTICS update_counter = ROW_COUNT;
  ...
```


Feedback & Error Handling

- **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*'
    - 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
      - MESSGE_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

```
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 specified:
    - 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
    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
 ...
 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):

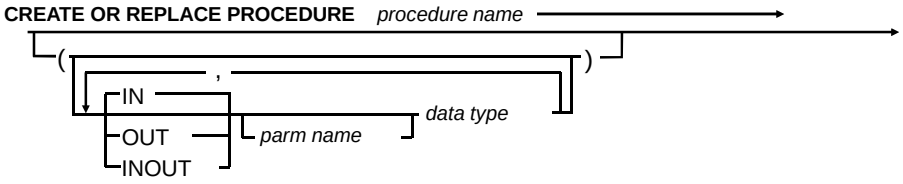
- **Static** – Things you know about during the procedure creation
- **Dynamic** – to handle things that can vary

Static and Dynamic mix example

```
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;
```

Routine Signatures

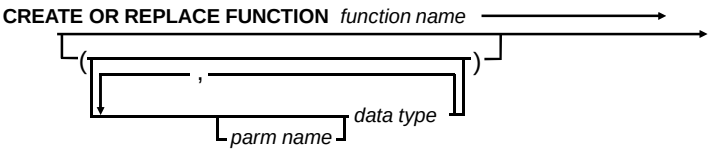
Create Procedure options



- 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

Create Function options



- Function name + number of parameters + parameter (family) data type make a unique signature
  - Enables overloading / polymorphism for functions within the same library

    Ex. These two functions are allowed in the same library (mylib):

    mylib.func1(parm1 int)
    mylib.func1(parm1 char)

Note: casting rules have changed (softened) as of v7r2

Stored Procedure Result Sets

Result Sets & Procedures

- 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

SQL Procedures - Result Sets

```
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

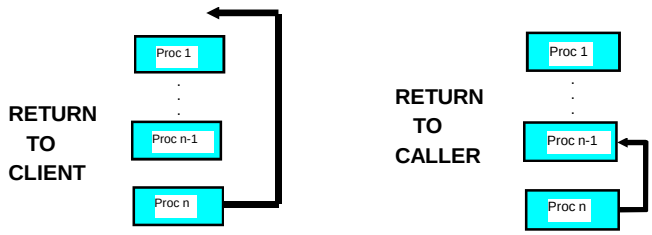
Result Set Consumer Control

- 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

```
...
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;
...
```

PROJNAME	PRSTAFF
GENERAL ADMIN ...	6.00
PAYROLL PROGR...	2.00
PERSONNEL PRO...	1.00
ACCOUNT PROGR...	2.00

Easy Integration!

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

Programming considerations

SQL Routines - Compound Statement

- Compound statements can be executed as standalone dynamic SQL requests within SQL scripts
  - Conditional logic and error handling
  - Input values with global variables

```
BEGIN
 DECLARE CONTINUE HANDLER FOR SQLSTATE '42704'
   BEGIN  /* Table may or may not already exist*/
     INSERT INTO error_log
       VALUES(SESSION_USER, CURRENT TIMESTAMP, '42704');
   END;

  IF SESSION_USER <>  ' DBADMIN' THEN
    SIGNAL SQLSTATE '38001' SET MESSAGE_TEXT='Unauthorized User';
  END IF;

  IF Setuplib.Create_Var='YES' THEN
    DROP TABLE orders;
    CREATE TABLE orders(ordID CHAR(6), ordQty INTEGER, ordCustID CHAR(5));
  END IF;
END;
```

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 embedded Static SQL statements
- 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:

```
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
aacxW8plW8FjG8pnG8VzG8FD68Fj68:Hl8:dY_pB2qpdW8pdW8pdW_praqe
baqebaGEMj_vsPBs5bOJUUqnHVayEl_ogAlGWqz2jJCIE1dQEjt33hd5Sps5
cYGViD1urv7vGKeOcC4CwpCibb
```


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');
```

```
SELECT ROUTINE_DEFINITION FROM QSYS2.SYSROUTINE WHERE ROUTINE_NAME =
ROUTINE_DEFINITION
WRAPPED QSQ07010 aacxW8plW8FjG8prG8FLG8Fv68Vv69prl8VF38Fx2qpdW8pdW8pdX8phJ2u
```

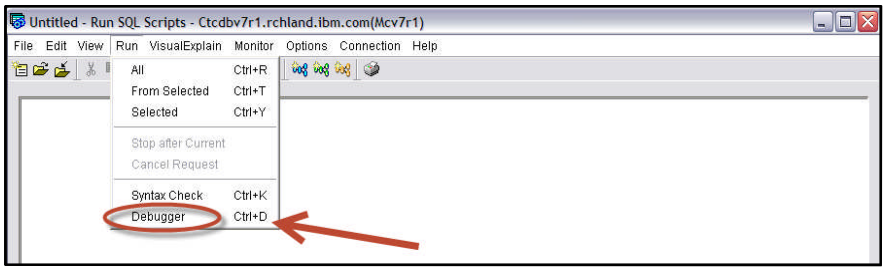
* And triggers in v7.2

Tools Available For Debugging

- Graphical Debuggers
  - IBM i Graphical Debugger
    - Not just for SQL Routines, any IBM i program
    - Part of IBM Toolbox for Java
  - IBM Data Studio
  - *Graphical debug white paper: https://ibm.biz/Bdxpni*
- STRDBG command

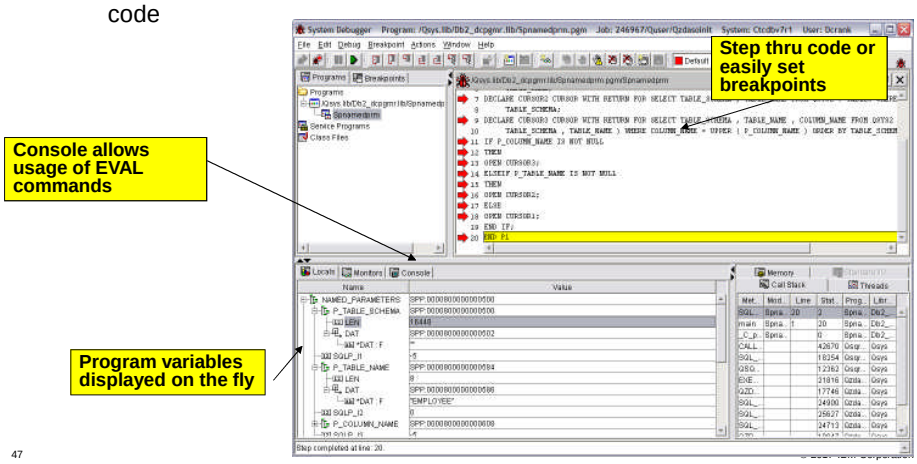
Run SQL Scripts Debugger Interface

- Powerful tool for debugging SQL routines
- Activate two ways
  - Debugger option from Run menu pull-down
  - Use Ctrl+D hotkey sequence



IBM i Graphical Debugger & SQL Routines

- Display, modify and monitor variables
- Set breakpoints
- Step buttons may need to be clicked multiple times
  - Single SQL statement may be implemented with multiple lines of C code



IBM i Debuggers – Enablement Steps

- Use SET OPTION clause to enable SQL source-level debug
  SET OPTION DBGVIEW = *SOURCE
- Debuggers only support 10 character program names
  - Use SPECIFIC clause to provide meaningful short name

  CREATE PROCEDURE **LONG_PROCEDURE_NAME** ( )
  SPECIFIC **LONGPROC** …

- Specify BEGIN label to enable EVAL command for local variables

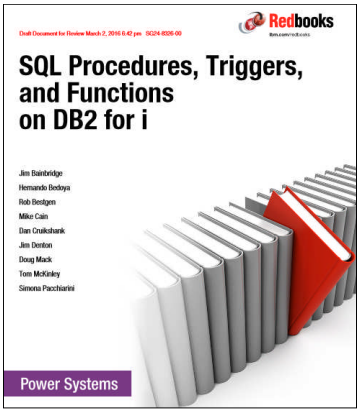
```
CREATE PROCEDURE proc1(IN p1 INT)
  LANGUAGE SQL   SET OPTION DBGVIEW=*SOURCE
sp: BEGIN
  DECLARE x INT;
  SET x = p1 + 5;
END;
```

More Information

The Full Story

A refreshed (2016) procedure RedBook is available!

http://www.redbooks.ibm.com/redpieces/abstracts/sg248326.html



Additional Information

- **DB2 for i Websites**
  - Homepage: www.ibm.com/systems/power/software/i
  - Technology Updates
    www.ibm.com/developerworks/ibmi/techupdates/db2
  - developerWorks Zone: www.ibm.com/developerworks/data/products.html
- **Forums**
  - developerWorks:
    https://ibm.com/developerworks/forums/forum.jspa?forumID=292
- Articles on procedure resolution related to default parameters
  - http://www.ibm.com/developerworks/ibmi/library/i-sqlnaming/index.html
  - http://www.ibm.com/developerworks/ibmi/library/i-system_sql2/index.html

DB2 for IBM i Lab Services

- **Facilitated workshops covering current state, requirements, future state, possible solutions, implementation best practices, and formulation of a strategic roadmap:**
  - **RCAC**
  - **Temporal Tables**
- **Customized consulting workshops**
  - **Advanced SQL and Datacentric Programming**
  - **SQL Performance Best Practices, Monitoring and Tuning**
- **Consulting on any DB2 for i topic**

For more information, contact mcain@us.ibm.com

Trademarks and Disclaimers