



IBM Systems Group

SQL Procedural Language SQL Stored Procedures, Triggers, and Functions

Alison Butterill, IBM Canada



COMMON Belgium Congress
17 November 2004

SQL Procedural Language

- Benefits of SQL Procedural Language
 - Supports rapid development of stored procedures, triggers, and User Defined Functions
 - Enhances portability
 - Supported across DB2 UDB Family
 - Similar to procedure languages available from other DBMS (PL/SQL, T-SQL, etc)
 - DB2 UDB implementation not proprietary, follows SQL PSM Standard
 - Makes it easier for SQL programmers to be productive faster on the iSeries
- Prerequisites on the development system
 - Openness Includes (5722-SS1) required on development system
 - DB2 SQL Development Kit requirement **eliminated in V5R2**
 - C compiler requirement **eliminated in V5R1**

Basic Programming Constructs

Program Control Operations within SQL

- CASE
- IF
- FOR
- LOOP
- REPEAT
- Declaring Variables
- Error Handling

```
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;
```

```
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;
```

New SQL Statements

- Conditional Logic, Declaration of Variables, Etc
 - BEGIN and END
 - DECLARE (local variables)
 - SET (local variables)
 - Comments
 - CASE (two forms), END CASE
 - IF, THEN, ELSE, END IF
 - FOR, END FOR, LOOP, END LOOP
 - LEAVE (loop or block)
 - REPEAT, END REPEAT
 - WHILE, END WHILE
 - GET DIAGNOSTICS (SQLCA-like information)
 - CALL - SQL procedure, External Procedures like HLL
 - Normal DDL and DML

Variable Declaration

DECLARE \downarrow *SQL Variable* *data type* $\left\{ \begin{array}{l} \text{DEFAULT NULL} \\ \text{DEFAULT constant} \end{array} \right.$;

- Variable initialized when the SQL 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

Conditional Constructs

- CASE Expression - two flavors...
 - First form:

```
CASE workdept
  WHEN 'A00' THEN UPDATE department
    SET deptname = 'ACCOUNTING';
  WHEN 'B01' THEN UPDATE department
    SET deptname = 'SHIPPING';
  WHEN 'A01' THEN UPDATE department
    SET deptname = 'MARKETING';
  ELSE UPDATE department
    SET deptname = 'UNKNOWN';
END CASE
```

Conditional Constructs

- Second form:

```
CASE
  WHEN vardept='A00' THEN UPDATE department
    SET deptname = 'ACCOUNTING';
  WHEN vardept='B01' THEN UPDATE department
    SET deptname = 'SHIPPING';
  WHEN vardept='A01' THEN UPDATE department
    SET deptname = 'MARKETING';
  ELSE UPDATE department
    SET deptname = 'UNKNOWN';
END CASE
```

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;
```

Looping Constructs

- For statement - execute a statement for each row of a table

```
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

- LOOP Statement - repeat the execution of a statement

```
fetch_loop:
LOOP
  FETCH cursor1 INTO
    v_firstname, v_midinit, v_lastnm;
  IF SQLCODE <> 0 THEN
    LEAVE fetch_loop;
  END IF;
  SET fullname = v_firstname || ' ' || v_midinit || ' ' || v_lastnm;
END LOOP;
```

- LEAVE statement continues execution by leaving the specified loop or block
 - Can be used with any of the looping constructs (except FOR loop)

Looping Constructs

- Repeat Statement - similar to Loop except for loop exit condition supported

```
REPEAT
  FETCH cursor I INTO
    v_firstname, v_midinit, v_lastname;

  SET fullname = v_firstname || ' ' || v_midinit || ' ' || v_lastname;

UNTIL SQLCODE <> 0
END REPEAT
```

Looping Constructs

- While Statement - same as REPEAT but exit condition checked before loop entry

```
WHILE at_end=0 DO
  FETCH cursor I INTO
    v_firstname, v_midinit, v_lastname;

  SET fullname = v_firstname || ' ' || v_midinit || ' ' || v_lastname;

  IF SQLCODE <> 0 THEN
    SET at_end=1;
  END IF;
END WHILE;
```

Looping Constructs

- **ITERATE Statement** - causes flow of control to return to the beginning of labeled loop
 - Can be used with any loop type
 - New with V5R2

```
ins_loop: LOOP
```

```
    FETCH c1 INTO v_dept,v_deptname,v_admdept;
```

```
    IF at_end = 1 THEN LEAVE ins_loop;  
    ELSEIF v_dept = 'D11' THEN ITERATE ins_loop;  
    END IF ;
```

```
    INSERT INTO department VALUES ('NEW ',v_deptname,v_admdept);  
END LOOP;
```

GOTO statement

- **GOTO statement** - included primarily for error handling

```
IF P1 = 1 THEN  
    GOTO WHILELOOP2;  
END IF;
```

```
...
```

```
WHILELOOP2: WHILE at_end=0 DO  
    FETCH cursor1 INTO  
        v_firstname, v_midinit, v_lastname;
```

```
    SET fullname = v_firstname || ' ' || v_midinit || ' ' || v_lastname;  
END WHILE;
```

```
...
```

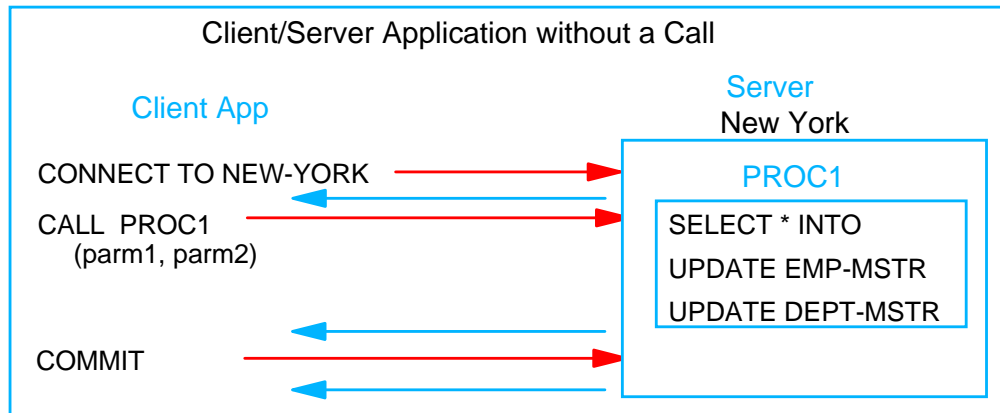
SQL Stored Procedures

What is a Stored Procedure?

- **Just a called program**
 - Called from SQL-based interfaces via SQL CALL statement
- **Supports input and output parameters**
 - Result sets on some interfaces
- **Follows security model of iSeries**
 - Enables you to secure your data
 - iSeries adopted authority model can be leveraged
- **Useful for moving host-centric applications to distributed applications**

Stored Procedures...

- Improved Client/Server (DRDA) Performance
 - CALL of stored procedure results in a significant reduction in conversation between client and server
 - SQL statements are embedded in (compiled) program on server resulting in more efficient performance of SQL



Recipe for a Stored Procedure...

1. **Create SQL stored procedure (one-time operation) using any SQL interface**

```

CREATE PROCEDURE total_val (IN Member# CHAR(5),
                           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. **Start calling procedure (total_val) from any SQL-based interface that supports SQL CALL statement**

SQL Stored Procedures

- SQL Procedure Support (since V4R2):

```

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 UDB for iSeries uses the SQL SP to create a C program object with the CRTSQLCI and CRTPGM commands

SQL Procedure Body

- Compound statement - specify a statement that groups other statements together in an SQL procedure
BEGIN ATOMIC or **NOT ATOMIC**
 SQL procedure statement; [repeatable]
END
- With V5R2, compound statements can be nested within each other
- **ATOMIC** indicates that if an error occurs, all SQL statements in the compound statement group will be rolled back.
 - If **ATOMIC** specified, **COMMIT** or **ROLLBACK** cannot be specified in the Stored Procedure
 - Starting with V5R2, **ATOMIC** procedures must also be created with **COMMIT ON RETURN YES** - any existing **ATOMIC** procedures will have to be changed if recreated on a V5R2 system
- **NOT ATOMIC** indicates that an error does **NOT** cause statements to be rolled back

SQL Procedure Body

- Statements must be ordered as follows:

BEGIN

<local variable declarations>

<local cursor declarations>

<local handler declarations>

<SQL statement list/procedure logic>

END

Basic Constructs

- Assignment statement - for assigning a value to SQL output parameter or SQL variable

```
SET total_salary = emp_salary + emp_commission;  
SET total_salary = NULL;  
SET loc_avgsalary =  
    (SELECT AVG(salary) FROM employees); (V4R4)
```

- Comments - two options
 - Two consecutive hyphens (--)
 - Bracketed comments (/* ... */)

Basic Constructs

- Call statement - for invoking stored procedures

CALL ProcedureName(Parm1, Parm2, etc);

- ▶ Up to 253 arguments allowed on CALL statement
- ▶ A parameter can contain SQL parameter, SQL variable, constant, special register, or NULL

- Provides a mechanism for accessing system functions and APIs from an SQL Stored Procedure

Error Handling

- No direct access to SQLCA provided
 - Can access error information by declaring SQLSTATE or SQLCODE variables that DB2 UDB will automatically update
 - Sample usage:
DECLARE SQLSTATE CHAR(5);
DECLARE SQLCODE INTEGER;

DELETE FROM tablex WHERE col1=100;
IF SQLSTATE='02000' THEN
- GET DIAGNOSTICS EXCEPTION... also provides access to some of the SQLCA fields
- **NOTE:** Every procedural statement is an SQL statement, potentially need to save SQLSTATE/SQLCODE after every statement

Getting Feedback

- **GET DIAGNOSTICS** statement

- Lack of data structure support results in no SQLCA access from an SQL Procedure, GET DIAGNOSTICS purpose is to provide some of this information
- Current support just allows you to retrieve the number of rows affected count from an Insert, Update, or Delete Statement

```
DECLARE update_counter INTEGER;  
...  
UPDATE...  
GET DIAGNOSTICS update_counter = ROW_COUNT;  
...
```

Error Handling

- Error handling can be done with conditions and handlers:

```
DECLARE row_not_fnd CONDITION FOR '02000';  
--SQL Condition declare for SQLSTATE associated  
--with no rows meeting criteria error
```

```
DECLARE CONTINUE HANDLER FOR row_not_fnd  
SET at_end='Y';  
--Tell database to assign 'Y' to at_end and continue  
--processing when row_not_fnd condition raised
```

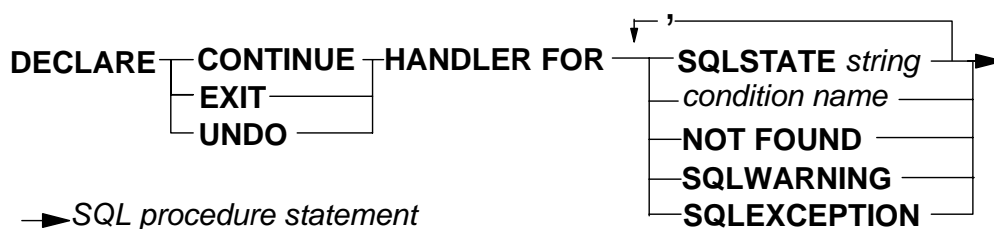
```
DELETE FROM tablex WHERE hiredate>='01/01/2001';  
--Handler would continue processing on statement  
--following this failing statement
```

ERROR HANDLING - Condition declaration

DECLARE *condition name* **CONDITION FOR** *string constant*;

- Condition name allows user friendly alias (eg, row_not_found) to be associated with more-cryptic SQLSTATE values like '02000'
- Condition name must be unique within the Stored Procedure

Error Handling - Handler declaration

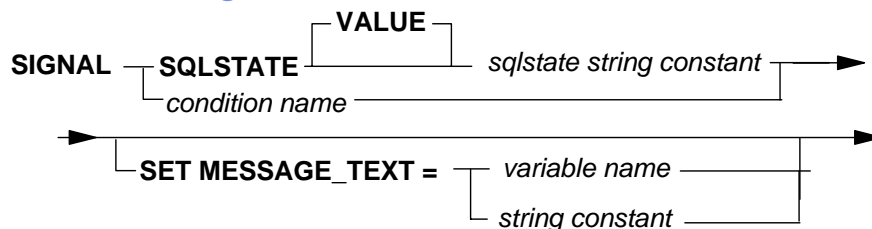


- Error handler associated with an exception(s) or completion condition(s) for the procedure
 - ▶ Starting with V5R2, handler can be associated with multiple conditions AND... handler can execute a compound statement (instead of using LOOP workaround)
- User specifies statement for handler to execute as well as how processing control is resumed after the error has been handled (CONTINUE, EXIT, UNDO)
 - ▶ If no handler for an error, then error returned to invoker and processing ends

Error Handling - Handler Declaration

- UNDO
 - ▶ ROLLBACK the changes made by the compound statement and invoke the handler. Once the handler is invoked successfully, control is returned to the end of the compound statement.
 - ▶ Must be an ATOMIC compound statement/procedure
- CONTINUE
 - ▶ Once the handler completes, control is returned to the SQL statement following the one that raised the exception
- EXIT
 - ▶ Once the handler completes, control is returned to the end of the procedure

Error Handling - SIGNAL statement



- SIGNAL statement causes error or warning condition to be returned with the specified SQLSTATE & optional message text
 - Message text can be up to 70 bytes in length, longer messages will be truncated without warning
 - *Diagnostic string* - an expression with a type of CHAR or VARCHAR that describes the error/warning
 - Text copied into the SQLCA for the procedure and the invoker (depending on interface)
 - **EXAMPLE:** VB program would retrieve the user-defined SQLSTATE and message text via the Connection object (Conn.Error(i).SQLSTATE & Conn.Error(i).Description)
 - If invoked from a handler, it does NOT cause an infinite loop
- ODBC & JDBC drivers enhanced in V5R1 so that this error information is available to the client application

Error Handling - SIGNAL statement

■ More SIGNAL details...

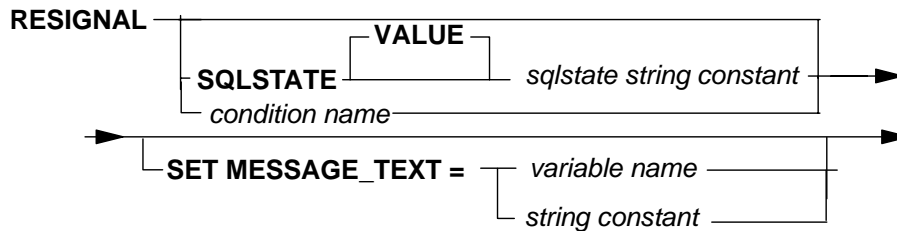
- If SQLSTATE class is '01' or '02', warning is signaled and the SQLCODE is set to +438
 - ▶ Otherwise SQLCODE set to -438
 - ▶ Negative SQLCODE cause output variables to NOT be returned
 - ▶ SystemMessage for SQLCODE 438 does NOT exist, just a code for program to program communications
- If a handler for the signalled exception exists, exception is handled and control transferred to handler
 - ▶ Otherwise, control returned immediately to the invoker for exceptions OR to the next statement for warning (01 & 02 class)
- Recommendation: Define your own SQLSTATEs based on the ranges reserved for applications

Error Handling - SIGNAL statement

■ Defining application/user SQLSTATE values for Procedures (& Functions)

- SQLSTATE breakout: XXYZZ (XX = Class, Y = Subclass)
- Do not use SQLSTATE with '00' Class
- Can define any SQLSTATE class that begin with '7'-'9' & 'I' thru 'Z'
 - ▶ These class values will be handled by SQLEXCEPTION handler (any class except '00', '01', '02')
- SQLSTATE classes that begin with '0'-'6' or 'A'-'H' are reserved for database manager. Subclasses '0'-'H' are also reserved.
 - ▶ Applications can use Subclass values 'I'-'Z'
 - ▶ The unreserved subclasses allow applications to define their own Warning or Not Found states (eg, '01PGM' & '02PGM') - would be handled by the WARNING or NOTFOUND handlers

Error Handling - SIGNAL statement



- Similar to SIGNAL, except that RESIGNAL can only be used in a handler to resignal an error or warning
 - Don't need to specify a State or Message just: **RESIGNAL;**
 - In this case, the invoker is returned the original condition that caused the handler to be invoked
 - Have the option of supplying your own SQLSTATE or diagnostic message text.

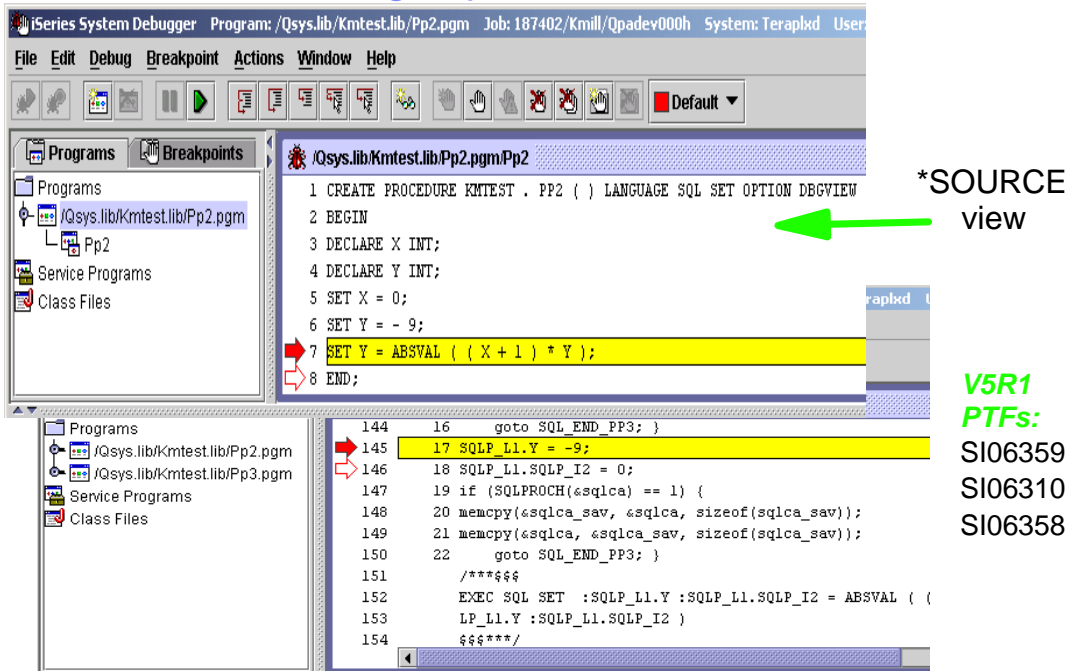
Example:

- Override system message for SQLSTATE 02000 "Row not found" with "Specified part not found"
- Invoker would be able to access overwritten message in SQLCA

Tool Considerations

- **iSeries Navigator**
 - A simple graphical editor for developing SQL Procedures
 - Runtime debug by displaying contents of result sets and output parameters (new in V5R2)
- **DB2 Stored Procedure builder (DB2 Development Center)**
 - More advanced graphical editor
 - Embedded in DB2 Personal Developer's Edition, which is available for download at: ibm.com/software/data/db2/udb/downloads.html
 - Need to setup a DRDA connection to with the DB2 Connect component that's part of DB2 Personal Developer Edition
 - Also need to setup a JDBC connection with the iSeries Toolbox JDBC driver

SQL Procedure Debug Improvements



***SOURCE view**

```

1 CREATE PROCEDURE KMTEST . PP2 ( ) LANGUAGE SQL SET OPTION DBGVIEW
2 BEGIN
3 DECLARE X INT;
4 DECLARE Y INT;
5 SET X = 0;
6 SET Y = - 9;
7 SET Y = ABSVAL ( { X + 1 } * Y );
8 END;

```

V5R1 PTFs:
SI06359
SI06310
SI06358

```

144 16 goto SQL_END_PP3; }
145 17 SQLP_L1.Y = -9;
146 18 SQLP_L1.SQLP_I2 = 0;
147 19 if (SQLPROCH(sqlca) == 1) {
148 20 memcpy(sqlca_sav, sqlca, sizeof(sqlca_sav));
149 21 memcpy(sqlca, sqlca_sav, sizeof(sqlca_sav));
150 22 goto SQL_END_PP3; }
151 /**$$$
152 EXEC SQL SET :SQLP_L1.Y :SQLP_L1.SQLP_I2 = ABSVAL ( (
153 LP_L1.Y :SQLP_L1.SQLP_I2 )
154 $$$***/

```

SQL Procedure Debug Improvements - *SOURCE tips

- Accessing SQL variables & parameters

Parameters:

EVAL P22.PARM1

Variables:

EVAL SP

EVAL SP.X

EVAL *SP.Z :S 5

```
CREATE PROCEDURE p22(IN parm1 INTEGER)
```

```
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;
```

iSeries Navigator - Improved Error Highlighting

- **SQL Script Center will now highlight keyword causing syntax error -**

- ▶ Very similar to the error feedback in RUNSQLSTM spool files

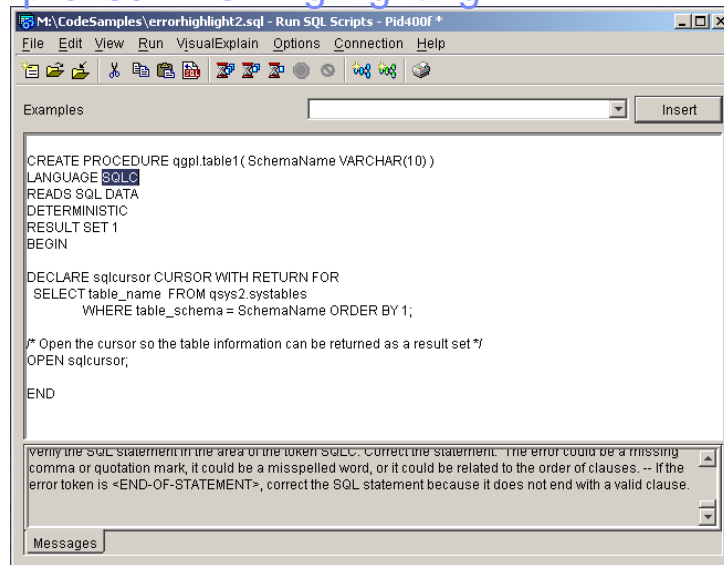
- ▶ V5R1 PTF Available

- Scheduled for next CA Service Pack (12/6/02)

OR

- Download from:

<ftp://ftp.software.ibm.com/as400/products/clientaccess/win32/v5r1m0/files>



Considerations

- Dynamic SQL is allowed in the SQL Stored Procedure
- CONNECT for DRDA-type processing also allowed in the SQL Stored Procedure
- Prior to V5R2, debugging had to be done on the C Program listing instead of at the SQL statement source level
 - Starting with V5R1 it is create debuggable version of SQL Procedure on any interface (not possible with Ops Navigaator in previous releases) by embedding the following statement:
SET OPTION DBGVIEW = *STMT
 - V5R2 brings *SOURCE debuggable view...

Catalog Considerations

- SYSPROCS & SYSPARMS catalogs updated when an SQL Stored Procedure is created
- SQL_DATA_ACCESS & ROUTINE_DEFINITION columns part of SYSROUTINES
 - ROUTINE_DEFINITION will contain the SQL procedure body
 - SQL_DATA_ACCESS will contain 'NO SQL', 'READS SQL DATA', etc values

Transaction Considerations

- ILE C program object for Procedure created with Activation Group *CALLER
- If SQL procedure created as ATOMIC then the invoker has to be at a Commit boundary before invoking the ATOMIC SQL Stored Procedure
- COMMIT and ROLLBACK not allowed in a procedure with the ATOMIC attribute

Moving Procedures into Production (V5R1 & beyond)

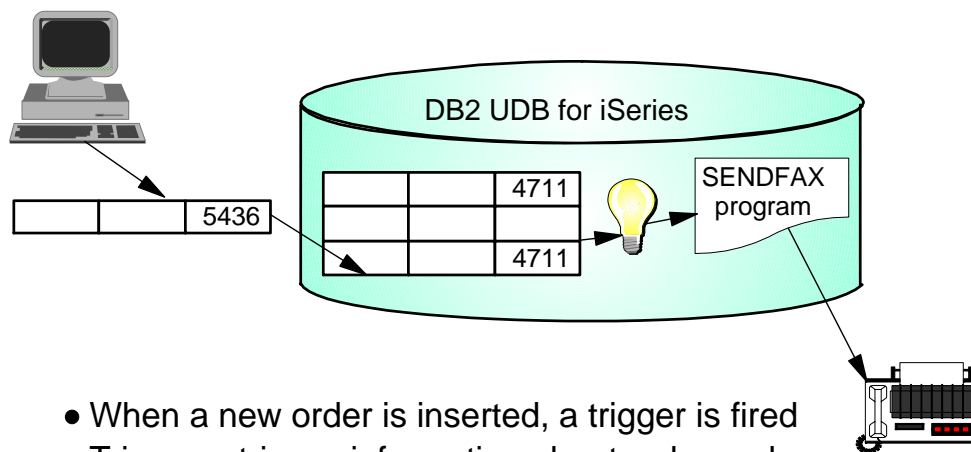
- DB2 UDB does try to automatically recognize the C program on the restore as an SQL Stored Procedure, but there are exceptions....
 - ▶ If DB2 UDB does not find a matching procedure in the catalogs, then the C program is registered as an SQL Stored Procedure
 - ▶ If DB2 UDB finds **one** procedure with 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 UDB finds one or more procedure 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 (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

SQL Triggers

Triggers: Introduction

- Triggers are user-written programs
 - associated with a physical file
 - activated by DB2 UDB for iSeries before or after a database change
 - independent from applications
 - can be developed with any supported compiler
- When do you need triggers?
 - to consistently enforce complex business rules
 - implement special application requirements
 - to monitor critical files
 - validate data
 - in a client-server environment

Triggers: An Example



- When a new order is inserted, a trigger is fired
- Trigger retrieves information about order and customer
- A confirmation fax is automatically sent

OS/400 Support for Triggers

- V4R5 and Prior Releases of OS/400
 - ▶ External triggers only (aka Native or System triggers)
 - ▶ Two commands
 - ADDPFTRG
 - RMVPFTRG
 - ▶ External triggers can be added or removed from a file using the Database function in Operations Navigator
 - ▶ A file can have a total of 6 triggers
 - ▶ No triggers on catalog files or tables
- V5R1 - Two Types of Triggers
 - ▶ External Triggers (see above)
 - First available with DB2/400 in V3R1
 - Originally referred to as Native Triggers
 - ▶ SQL Triggers
 - New in V5R1

V5R1 Trigger Enhancements

- SQL Triggers
 - Column level
 - Row level
 - Statement level
- More than 1 trigger per database event
 - maximum 300 per physical file or table
 - triggers for same event, fired in the order created
 - identified or qualified by Trigger name
- CHGPFTRG command
 - disable an active or enabled trigger
 - enable an inactive or disabled trigger
- 'Read only' Trigger
 - use carefully

SQL Trigger Components

- Base file or table
 - ▶ Physical file or table which the trigger is added to
- Trigger name
 - ▶ Provides unique trigger identification within a library
- Trigger event
 - ▶ The condition that causes the trigger to fire
 - Insert of a new row
 - Update of existing
 - Row
 - Column (Column level triggers)
 - Delete of an existing record
- Trigger time
 - ▶ When trigger program is to be run
 - ▶ Before or after the trigger event

SQL Trigger Components...

- Trigger Granularity
 - ▶ Column level triggers
 - Extension of UPDATE trigger event
 - Columns listed as part of UPDATE trigger event
 - UPDATE OF column_name_1, column_name_2, ...
 - Only update of a listed column causes trigger to fire
 - If no columns listed, update to any column causes trigger to fire
 - ▶ Row level triggers
 - FOR EACH ROW
 - Triggered action executed for each row satisfying trigger condition
 - If trigger condition never satisfied, triggered action never executed
 - ▶ Statement level triggers
 - FOR EACH STATEMENT
 - Triggered action executed only once for the event causing the trigger to fire regardless of the number of rows processed
 - If trigger condition never satisfied, triggered action executed once at end of statement processing
 - Not valid with Before triggers or Trigger Mode of DB2ROW

SQL Trigger Components...

- Trigger Mode
 - ▶ MODE DB2ROW
 - Trigger fires after each row operation
 - Only valid with Row level triggers
 - Exclusive function of DB2 UDB for iSeries
 - Not available in other DB2 UDB implementations
 - ▶ MODE DB2SQL
 - Trigger fires after all row operations are complete
 - If specified on a row level trigger, triggered action executed N times after all row operations completed
 - N = number of rows processed
 - Not as efficient as DB2ROW since each row is processed twice
 - Only valid with After triggers

SQL Trigger Components...

- Transition Variables
 - ▶ aka Correlation Variables
 - ▶ Provides function similar to before and after images in trigger buffer for external triggers
 - ▶ Qualification of column names for the single row image before and/or after the trigger event has completed
 - OLD ROW - Before image of row
 - NEW ROW - After image of row
 - REFERENCING OLD ROW AS oldrow
REFERENCING NEW ROW AS newrow
 - ...newrow.salary > oldrow.salary + 10000...
 - ▶ Not valid with Statement level triggers

SQL Trigger Components...

- Transition Tables
 - ▶ Provides function similar to before and after images in trigger buffer for external triggers
 - ▶ A single SQL statement can process multiple rows
 - ▶ Temporary tables that contain the image of all affected rows before and/or after the trigger event completes
 - OLD TABLE - Before image of all affected rows
 - NEW TABLE - After image of all affected rows
 - REFERENCING OLD TABLE AS oldtbl
 - ...(SELECT COUNT(*) FROM oldtbl)...
 - ▶ Not valid with Before triggers or Trigger Mode of DB2ROW

SQL Trigger Components...

- triggered Action
 - ▶ Analogous to trigger program in external triggers
 - ▶ Three parts
 - SET OPTION
 - Specifies the options that will be used to create the trigger
 - WHEN
 - Search condition or execution criteria for Trigger Body
 - Specifies when the SQL statements in Trigger Body will be executed
 - SQL Trigger Body
 - Single SQL statement
 - Multiple SQL statements delineated with BEGIN and END

SQL Trigger Examples

Row Level Trigger with Simple Trigger Body

```
CREATE TRIGGER audit_spending
  AFTER UPDATE ON expenses
  REFERENCING NEW ROW AS nw
  FOR EACH ROW MODE DB2ROW
  WHEN (nw.total_amount > 10000)
    INSERT INTO travel_audit
      VALUES(nw.empno, nw.deptno, nw.total_amount,
             nw.end_date);
```

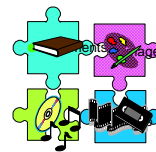
SQL Trigger Examples ...

Row Level Trigger with Complex Trigger Body

```
CREATE TRIGGER big_spenders
  AFTER INSERT ON expenses
  REFERENCING NEW ROW AS n
  FOR EACH ROW
  MODE DB2ROW
  WHEN (n.totalamount > 10000)
  BEGIN
    DECLARE emplname CHAR(30);
    SET emplname = (SELECT Iname FROM employee
                   WHERE empid = n.empno);
    INSERT INTO travel_audit
      VALUES(n.empno, emplname, n.deptno, n.totalamount, n.enddate);
  END
```

SQL User Defined Functions

User-Defined Functions (UDFs)



- Provides the ability to add application logic to the database
 - Function can be applied inside or outside the database
- Functions are called through SQL and are therefore reusable by all authorized database users, including programmers and end users
- Provide enhanced performance by:
 - Executing on the server rather than at the client
 - Being tightly integrated with database optimization and execution
- In combination with LOBs and UDTs, can provide processing methods for business objects tailored to specific business needs:
 - Advanced datatype support (e.g. multimedia, arrays, etc.)
 - Additional search technologies
 - contextual search, e.g. find all documents with the words DB2 and UDF
 - video scan e.g. find the scene starting with a sunset
- Use of standards allows functions to be easily added to the database

UDF Types

- External
 - Similar to external stored procedures
 - Use AS/400 high-level language to implement
- Sourced
 - Instead of defining from scratch, redefine an existing built-in function or UDF
 - UDT Strong Typing requires their own basic functions
- SQL
 - All the function logic coded with SQL
 - Utilizes procedural SQL introduced in V4R2

SQL UDFs

- "Pure" SQL implementation
 - Defined, written, and registered using CREATE FUNCTION statement
- Easier to write than External Functions
 - Parameter passing rules defined by the database support

```
CREATE FUNCTION slice( p1 VARCHAR(256) RETURNS CHAR(4)  
LANGUAGE SQL
```

```
-- returns the first two and the last two characters of the Varchar
```

```
s1: BEGIN
```

```
  DECLARE temp CHAR(4);
```

```
  SET temp = CONCAT(SUBSTR(p1,1,2), SUBSTR(p1,LENGTH(p1)-1,2));
```

```
  RETURN temp;
```

```
END s1;
```

```
SELECT slice(NAME) FROM Employee;
```

Resolving UDFs

- Finding the right UDF to call can be challenging for DB2 when you consider that:
 - UDFs support function overloading
 - UDFs can be sourced from existing functions
 - UDFs can have the same name as built-in functions
 - Unqualified UDF invocations allowed
- Three determining resolution factors are:
 - Function Signature
 - Function Path
 - Parameter Matching

Function Signature

- Made up of Collection, Name, and Parameter Type(s)
 - Allows for multiple functions to have the same name in the same collection/library/schema
 - Return value NOT part of the signature
 - Parameter length, precision & scale are NOT used
- Signature must be unique within a collection
- Signature examples:
 - mylib.myUDF(integer, char(5)) & mylib.myUDF(integer, integer) can exist in the same collection
 - xyz(integer, char(50)) & xyz(integer, char(5)) CANNOT exist in the same collection

Function Path

- Ordered list of collection names
 - Used when searching for unqualified data type references and function & procedure invocations
 - For System Naming (*SYS), default path is *LIBL
 - For SQL Naming (*SQL) default path is: QSYS, QSYS2, <user-id>
- Path can be changed with SET PATH statement
- QSYS & QSYS2 assumed to be first in path unless explicitly put in another position
 - SYSTEM PATH is equivalent to QSYS, QSYS2
- Examples:
 - SET PATH myfunc, SYSTEM PATH
(Result Path: myfunc, QSYS, QSYS2)
 - SET PATH myfunc, morefunc
(Result Path: QSYS, QSYS2, myfunc, morefunc)

Parameter Matching

- Done by parameter type only
- Exact match is best, DB2 chooses the next best promoted argument

SELECT UDF(charcolumn), col2 FROM Table1

Char Best to Worst Order



Char/Graphic, VarChar/VarChar, CLOB/DBCLOB

Function Signatures: ✓ Fred.UDF(varchar)
 Joshua.UDF(clob)
 Jenna.UDF(integer)

Notes:

A function is considered applicable to a given call if its function name matches the call and if the arguments of the call are promotable to the parameters of the function. This means that the data type of each function parameter either must match the data type of the corresponding call argument or must be found on the right hand side of the argument on the promotion path. Following table illustrates the available promotion paths:

Data Type	Type can be Promoted To: (listed best to worst)
char or graphic	char or graphic, varchar or vargraphic, clob or dbclob
varchar or vargraphic	varchar or vargraphic, clob or dbclob
integer	integer, smallint, decimal or numeric, real, double
smallint	smallint, integer, decimal or numeric, real, double
decimal or numeric	decimal or numeric, real, double
real	real, double
double	double, real

Coding UDFs - Considerations

- **UDFs should not perform operations that take a long time (minutes or hours)**
 - Invoked from a low-level in DB2 which holds resources (locks & seizures) for duration of the UDF execution
 - If UDF doesn't finish in allocated time, then the SQL statement fails (override with UDF_TIME_OUT option in query INI file)
- **Not a good idea for UDF to operate on same table(s) as invoking SQL statement**
 - Especially operations that conflict with the invoking statement
 - Avoid inserts, updates, and delete ops on the same table(s)

Additional Information

- **DB2 UDB for iSeries home page** - <http://www.iseries.ibm.com/db2>

- **Newsgroups**

- ▶ USENET: comp.sys.ibm.as400.misc, comp.databases.ibm-db2
- ▶ iSeries Network (NEWS/400 Magazine) SQL & DB2 Forum - <http://www.iseriesnetwork.com/Forums/main.cfm?CFApp=59>

- **Education Resources - Classroom & Online**

- ▶ http://www.iseries.ibm.com/db2/db2educ_m.htm
- ▶ <http://www.iseries.ibm.com/developer/education/ibo/index.html>

- **DB2 UDB for iSeries Publications**

- ▶ Online Manuals: <http://www.iseries.ibm.com/db2/books.htm>
- ▶ Porting Help: <http://www.iseries.ibm.com/developer/db2/porting.html>
- ▶ DB2 UDB for iSeries Redbooks (<http://ibm.com/redbooks>)
 - **Stored Procedures & Triggers on DB2 UDB for iSeries (SG24-6503)**
 - **DB2 UDB for AS/400 Object Relational Support (SG24-5409)**
 - **SQL Query Engine Redpiece**
<http://publib-b.boulder.ibm.com/Redbooks.nsf/RedpieceAbstracts/sg2456598.html>
- ▶ **SQL/400 Developer's Guide** by Paul Conte & Mike Cravitz
 - <http://www.iseriesnetwork.com/str/books/Uniquebook2.cfm?NextBook=183>
- ▶ **iSeries and AS/400 SQL at Work** by Howard Arner
 - <http://www.sqlthing.com/books.htm>