

DB2 Version 8 SQL Procedures

Quickstart Education

Maintained by Paul Yip (ypaul@ca.ibm.com)

February 2003

IBM Software Group

BEFORE YOU BEGIN

- This presentation is designed to be interactive with "quick labs" scattered throughout the presentation to re-enforce concepts
- Attendees should have the DB2 Development Center launched and be ready to perform the "quick labs".
- The DB2 SAMPLE database should be created and a supported C compiler should be installed and configured (discussed next)

Setting up the Build Environment

IBM Software Group

Environment Setup for SQL Procedures

- Install Application Development Client [on server](#)
- Install DB2 supported C/C++ compiler [on server](#)
- Configure compiler environment [if default not appropriate](#)
 - ▶ [DB2_SQLROUTINE_COMPILER_PATH](#) registry variable
- Configure compiler command [if default not appropriate](#)
 - ▶ [DB2_SQLROUTINE_COMPILE_COMMAND](#) registry variable
- REQUIRED: db2stop/db2start if variables changed

Default Compiler Configuration

- No setup is required if you are using the following:

- AIX
 - ▶ IBM VisualAge C++ 5.0
- Solaris
 - ▶ Forte C++ Version 5.0
- Linux
 - ▶ GNU/Linux g++
- HP-UX
 - ▶ HP aC++ Version A.03.31

- **Windows**
 - ▶ **Always requires compiler configuration**

Windows Compiler Configuration

- For Microsoft Visual C++ Version 5.0:
 - ▶ db2set DB2_SQLROUTINE_COMPILER_PATH="c:\devstudio\vc\bin\vcvars32.bat"
- For Microsoft Visual C++ Version 6.0:
 - ▶ db2set DB2_SQLROUTINE_COMPILER_PATH="c:\Micros~1\vc98\bin\vcvars32.bat"
- For Microsoft Visual C++ .NET
 - ▶ db2set DB2_SQLROUTINE_COMPILER_PATH="D:\Program Files\Microsoft Visual Studio .NET\Common7\Tools\vsvars32.bat"
- **Important:** ensure correct path to Visual Studio install path is correct

Configuration for non-default compilers (FYI)

- **AIX:** IBM C for AIX Version 5.0
 - ▶ `db2set DB2_SQLROUTINE_COMPILE_COMMAND=xlC_r -I$HOME/sqllib/include \`
`SQLROUTINE_FILENAME.c -bE:SQLROUTINE_FILENAME.exp \`
`-e SQLROUTINE_ENTRY -o SQLROUTINE_FILENAME -L$HOME/sqllib/lib -ldb2`
- **HP-UX:** C Compiler Version B.11.11.02
 - ▶ `db2set DB2_SQLROUTINE_COMPILE_COMMAND=cc +DAportable +ul -Aa +z`
`-D_POSIX_C_SOURCE=199506L \`
`-I$HOME/sqllib/include -c SQLROUTINE_FILENAME.c; \`
`ld -b -lpthread -o SQLROUTINE_FILENAME SQLROUTINE_FILENAME.o \`
`-L$HOME/sqllib/lib -ldb2`
- **Linux:** GNU/gcc
 - ▶ `db2set DB2_SQLROUTINE_COMPILE_COMMAND=cc -fpic -D_REENTRANT \`
`-I$HOME/sqllib/include SQLROUTINE_FILENAME.c \`
`-shared -lpthread -o SQLROUTINE_FILENAME -L$HOME/sqllib/lib -ldb2`
- **Solaris:** Forte C Version 5.0
 - ▶ `db2set DB2_SQLROUTINE_COMPILE_COMMAND=cc -xarch=v8plusa -Kpic -mt \`
`-I$HOME/sqllib/include SQLROUTINE_FILENAME.c \`
`-G -o SQLROUTINE_FILENAME -L$HOME/sqllib/lib \`
`-R$HOME/sqllib/lib -ldb2`

Optional: Specify Precompile Options

- Default is generally ok
- Customize precompile and bind options
 - ▶ `DB2_SQLROUTINE_PREOPTS`
- Only the following options are allowed:
 - ▶ `BLOCKING {UNAMBIG |ALL |NO}`
 - ▶ `DATETIME {DEF |USA |EUR |ISO |JIS |LOC}`
 - ▶ `DEGREE {1 |degree-of-parallelism |ANY}`
 - ▶ `DYNAMICRULES {BIND |RUN}`
 - ▶ `EXPLAIN {NO |YES |ALL}`
 - ▶ `EXPLSNAP {NO |YES |ALL}`
 - ▶ `INSERT {DEF |BUF}`
 - ▶ `ISOLATION {CS |RR |UR |RS |NC}`
 - ▶ `QUERYOPT optimization-level`
 - ▶ `SYNCPPOINT {ONEPHASE |TWOPHASE |NONE}`

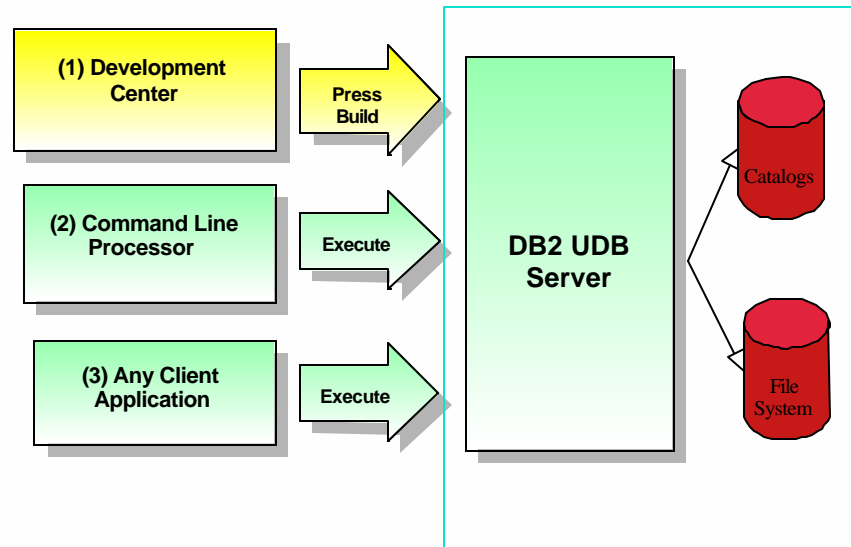
SQL Procedure Fundamentals

IBM Software Group

SQL Procedures

- Stored procedure in which procedural logic is contained in CREATE PROCEDURE statement
- Language similar to T-SQL, PL/SQL
- Subset of SQL/PSM standard
 - ▶ Database Language SQL - Part 4: Persistent Stored Modules (ISO/IEC 9075)
 - ▶ Include SQL procedures, SQL functions and Feature P01, "Stored modules" support

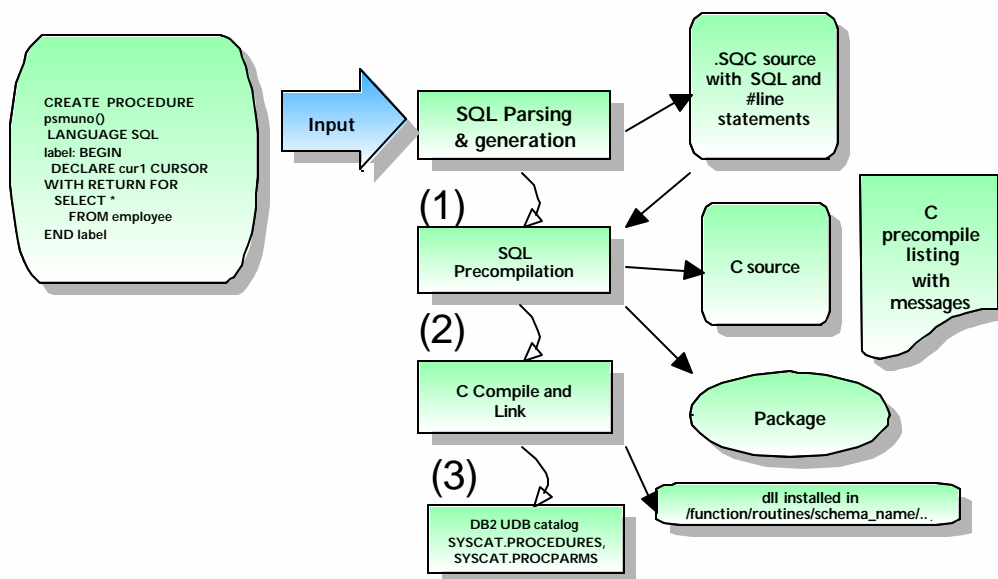
Creating SQL Procedures



DB2 Data Management Software



SQL Procedures - Behind the scenes



DB2 Data Management Software



QuickLab: Creating your first SQL procedure

- Using the development center (or command line), create the following procedure to validate your environment setup

```
CREATE PROCEDURE p1  
  BEGIN  
  END
```

CLP hint:

db2 connect to sample

db2 create procedure p1 begin end

Procedure Schema

- a procedure can be qualified or unqualified when using CREATE PROCEDURE
- Oracle Migration Recommendation: Convert package name to procedure schema name

p1 created in current schema



CREATE PROCEDURE p1 ...

CREATE PROCEDURE package1.p1...



p1 created in schema package1

Basic Procedure Structure

```
CREATE PROCEDURE proc [( {optional parameters} )]  
[optional procedure attributes]  
<statement>
```

<statement> is a **single** statement, or a set of statements grouped by BEGIN [ATOMIC] ... END

Optional Procedure Attributes

- LANGUAGE SQL
 - ▶ note: was required in V7.x
- RESULT SETS <n> (required if returning result sets)
- SPECIFIC my_unique_name
 - ▶ can be same as procedure name except limited to 18 chars
 - ▶ highly recommended for manageability:
 - GRANT EXECUTE ON SPECIFIC PROCEDURE ...
 - DROP SPECIFIC PROCEDURE ...
- [CONTAINS | READS | MODIFIES] SQL

Parameters

```
CREATE PROCEDURE proc(IN p1 INT, OUT p2 INT, INOUT p3 INT)
```

...

- IN - Input parameter
- OUT - Output parameter
- INOUT - Input and Output parameter
- All params must be provided in `CALL` statement
- In V7 only: All parameters must be variables (no literals or expressions) in `CALL`

NOTES:

Within the SQL procedure body, OUT parameters cannot be used on the LEFT side of an assignment or as a value in any expression. You can only assign values to OUT parameters using the assignment statement, or as the target variable in the INTO clause of SELECT, VALUES and FETCH statements. You cannot use IN parameters as the target of assignment or INTO clauses.

Comments

- -- This is an SQL-style comment
- /* This is a C-style comment */
(valid within SQL procedures)

QuickLab: CALL with IN, OUT, and INOUT parameters

- Create the following SQL Procedure:

```
CREATE PROCEDURE P1 ( IN    v_p1 INT,  
                     INOUT v_p2 INT,  
                     OUT   v_p3 INT)  
  
LANGUAGE SQL  
SPECIFIC P1  
BEGIN  
    -- my first sql procedure  
    SET v_p2 = v_p2 + v_p1;  
    SET v_p3 = v_p1;  
END
```

- Call the procedure from the CLP:

CALL P1 (10, 10, ?)

Use parameter markers (?) for OUT parameters when invoking from the CLP

Compound Statement

Compound Statement

```
BEGIN [ATOMIC]  
    <declare variables>  
    <declare conditions>  
    <declare statements>  
    <declare cursors>  
    <declare handlers>  
  
    <logic >  
  
END
```

Optionally atomic

Declarations

Logic -
Can contain other
compound stmts

Notes:

Non-atomic compound statements can be nested.

Atomic compound statements cannot be nested.

Non-atomic compound statements can be nested within Atomic statements but not vice-versa.

Variable Declaration

```
DECLARE var_name <data type> [ DEFAULT value];
```

■ Note: Default value is NULL

■ Examples:

```
DECLARE temp1 SMALLINT DEFAULT 0;  
DECLARE temp2 INTEGER DEFAULT 10;  
DECLARE temp3 DECIMAL(10,2) DEFAULT 100.10;  
DECLARE temp4 REAL DEFAULT 10.1;  
DECLARE temp5 DOUBLE DEFAULT 10000.1001;  
DECLARE temp6 BIGINT DEFAULT 10000;  
DECLARE temp7 CHAR(10) DEFAULT 'yes';  
DECLARE temp8 VARCHAR(10) DEFAULT 'hello';  
DECLARE temp9 DATE DEFAULT '1998-12-25';  
DECLARE temp10 TIME DEFAULT '1:50 PM';  
DECLARE temp11 TIMESTAMP DEFAULT '2001-01-05-12.00.00';  
DECLARE temp12 CLOB(2G);  
DECLARE temp13 BLOB(2G);
```

Assignment Statement

- SET total = 100;
 - ▶ Same as VALUES (100) INTO total;
- SET total = NULL;
 - ▶ any variable can be set to NULL
- SET total = (select sum(c1) from T1);
 - ▶ Condition is raised if more than one row
- SET first_val = (select c1 from T1 fetch first 1 row only)
 - ▶ fetch only the first row from a table (V8 only)
- SET sch = CURRENT SCHEMA;

Variable Name Scoping

```
L1: BEGIN
  DECLARE a INT DEFAULT 100;
  BEGIN
    DECLARE a INT DEFAULT 200;

    SET a = a + L1.a;
    UPDATE T1 SET b = b * 1.1
      WHERE b = a; -- WARNING (if a is a column name)!!
  END;
END L1;
```

- Names resolve first to columns, then to variables.

Quicklab: Compound statements and variables

- Create the following procedure:

```
CREATE PROCEDURE package1.proc1 (OUT result VARCHAR(50))
SPECIFIC p1
BEGIN
  DECLARE v_ibmreqd VARCHAR(1);
  SELECT * INTO v_ibmreqd FROM SYSIBM.SYSDUMMY1;

  -- to use DECLARE in the middle of a procedure,
  -- you need a new BEGIN..END block

  BEGIN
    DECLARE v_schema varchar(50);
    SET v_schema = current schema;
    SET result = v_schema || ' - ' || v_ibmreqd;
  END;
END
```

Cursors

- cursor declaration and usage:

```
DECLARE <cursor name> CURSOR [WITH RETURN <return target>]
    <SELECT statement>;
OPEN <cursor name>;
FETCH <cursor name> INTO <variables>
CLOSE <cursor name>;
```

- result sets can also be directly returned to CLIENT or CALLER for processing
 - ▶ CLIENT: result set will return to client application
 - ▶ CALLER: result set is returned to client or stored procedure that made the call

Quicklab: Positioned Update/Delete

(positioned UPDATE is similar)

```
CREATE PROCEDURE fire_employee(IN mid CHAR(1))
LANGUAGE SQL
BEGIN
    DECLARE middle CHAR(1);
    DECLARE SQLCODE INT;
    DECLARE cur1 CURSOR FOR SELECT midinit FROM
employee;

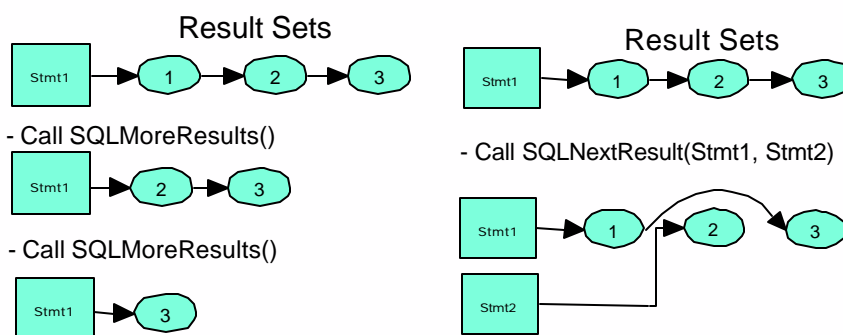
    OPEN cur1;
    FETCH cur1 INTO middle;
    WHILE (SQLCODE=0) DO
        IF (middle = mid) THEN
            -- commented out because we dont REALLY
            -- want to delete.. for illustration only
            -- DELETE FROM employee WHERE CURRENT OF
cur1;
        END IF;
        FETCH cur1 INTO middle;
    END WHILE;
END
```

QuickLab: Returning Result Sets

- Create and run the following stored procedure:

```
CREATE PROCEDURE set ()  
  DYNAMIC RESULT SETS 1  
  LANGUAGE SQL  
  BEGIN  
    DECLARE cur CURSOR WITH RETURN TO CLIENT  
      FOR SELECT name, dept, job  
        FROM staff  
        WHERE salary > 20000;  
    OPEN cur;  
  END
```

Multi ResultSet - SQLNextResult()



- New CLI function `SQLNextResult()`

Calling from CLI

```
SQLCHAR *stmt = (SQLCHAR *)      "CALL MEDIAN_RESULT_SET( ? )" ;

SQLDOUBLE  sal = 20000.0;          /* Bound to parameter marker in stmt */
SQLINTEGER salind = 0;             /* Indicator variable for sal */

sqlrc = SQLPrepare(hstmt, stmt, SQL_NTS);
sqlrc = SQLBindParameter(hstmt, 1, SQL_PARAM_OUTPUT,
                          SQL_C_DOUBLE, SQL_DOUBLE, 0, 0, &sal, 0, &salind);
SQLExecute(hstmt);

if (salind == SQL_NULL_DATA)
    printf("Median Salary = NULL\n");
else
    printf("Median Salary = %.2f\n", sal );

sqlrc = StmtResultPrint(hstmt);    /* Get first result set */
sqlrc = SQLMoreResults(hstmt);    /* Check for another result set */

if (sqlrc == SQL_SUCCESS) {      /* There is another result set */
    sqlrc = StmtResultPrint(hstmt);
}
```

DB2 Data Management Software

[See sqlib/samples/sqlproc/rsultset.c](http://www.ibm.com/sqlib/samples/sqlproc/rsultset.c)



Calling from JDBC

```
String procName = "TWO_RESULT_SETS";
String sql = "CALL " + procName + "(?, ?)";
CallableStatement callStmt = con.prepareCall(sql);

callStmt.setDouble (1, median);
callStmt.registerOutParameter (2, Types.INTEGER);

callStmt.execute();

int result = callStmt.getInt(2);

ResultSet rs = callStmt.getResultSet();
while (rs.next()) {
    ...
}
callStmt.getMoreResults();
rs = callStmt.getResultSet();
while (rs.next()) {
    ...
}
}
```

(2nd ResultSet)

DB2 Data Management Software

[See sqlib/samples/java/jdbc/spclient.java](http://www.ibm.com/sqlib/samples/java/jdbc/spclient.java)



Errors, Warnings and Condition Handling

IBM Software Group

SQLCODE and SQLSTATE

- Access requires explicit declaration:
 - `DECLARE SQLSTATE CHAR(5);`
 - `DECLARE SQLCODE INT;`
- Can be declared **ONLY** at outermost scope and automatically set by DB2 after each operation
- **SQLCODE**
 - = 0, successful.
 - > 0, successful with warning
 - < 0, unsuccessful
 - = 100, no data was found.
 - i.e. `FETCH` statement returned no data

SQLSTATE and conditions

- SQLSTATE
 - Success: SQLSTATE '00000'
 - Not found: SQLSTATE '02000'
 - Warning: SQLSTATE '01XXX'
 - Exception: Everything else
- A condition can be raised by any SQL statement
- General conditions:
 - SQLWARNING, SQLEXCEPTION, NOT FOUND
- Specific conditions:
 - SQLSTATE '01004'
- Can assign names to condition:
DECLARE trunc CONDITION FOR SQLSTATE '01004'

Condition Handling

- A condition handler must specify
 - handled conditions
 - where to resume execution (CONTINUE, EXIT or UNDO)
 - action to perform to handle the condition
- Action can be any statement (including control structures)
- Upon SQLEXCEPTION condition, if no handler exists, the procedure terminates and returns to the client with error

```
BEGIN [ATOMIC]
  DECLARE <type> HANDLER FOR <conditions>
    <handler-action>
  statement_1;
  statement_2;
  statement_3;
END
```

raises exception → statement_2;

CONTINUE point → statement_2;

UNDO or EXIT point → END

QuickLab: Condition Handling

- Create the following SQL procedure:

```
CREATE PROCEDURE Proc()
LANGUAGE SQL
BEGIN
  DECLARE at_end, truncated INT DEFAULT 0;
  DECLARE var1 VARCHAR(10);
  DECLARE trunc CONDITION FOR sqlstate '01004';
  DECLARE cur1 CURSOR FOR SELECT * FROM SYSIBM.SYSDUMMY1;

  DECLARE CONTINUE HANDLER FOR not found, SQLEXCEPTION
    SET at_end = 1;
  DECLARE CONTINUE HANDLER FOR trunc
    SET truncated = 1;

  OPEN cur1;
  WHILE at_end = 0 DO
    FETCH FROM cur1 INTO var1; -- Can raise trunc or not_found
  END WHILE;
  RETURN truncated;
END
```

Additional Notes

- Compound statement may contain any number of condition handlers
- Handler is searched from innermost to outermost block
- Inner declaration overrides outer (like in variable scoping)
- Handler for specific conditions have higher priority. E.g.:

```
DECLARE CONTINUE HANDLER FOR SQLWARNING ... ; -- (A)
DECLARE EXIT HANDLER FOR SQLSTATE '12345' ... ; -- (B)
...
SIGNAL SQLSTATE '12345'; -- Causes B to be executed
```

- SQL procedure terminates if no handler found for exception condition
- Handlers for warnings:
 - ▶ If handler is defined, it is executed and SQLCODE is cleared
 - ▶ Otherwise, execution continues, and SQLCODE is not cleared

Condition Handling

- SQLCODE and SQLSTATE values can only be accessed **by 1st statement** in condition handler
- If handler is successful, both SQLSTATE and SQLCODE **are cleared after handler finishes**

Example: Saving SQLSTATE & SQLCODE

```
CREATE PROCEDURE PSEUDOTSQ1()  
LANGUAGE SQL  
BEGIN  
    DECLARE SQLCODE, Saved_SQLCODE INT DEFAULT 0;  
    DECLARE CONTINUE HANDLER for SQLEXCEPTION  
        SET Saved_SQLCODE = SQLCODE;  
    -- SQLCODE is 0 at this point  
  
    UPDATE T1 SET b = b * 1.1 WHERE a = 100;  
    IF(Saved_SQLCODE <> 0)THEN  
        <code to handle the condition>  
    END IF;  
  
END
```

- Simple coding: "catch-all" handler
- But misses the point about condition handlers
i.e.: separating condition handling from the logic

Raising a condition explicitly (SIGNAL & RESIGNAL)

```
BEGIN
  DECLARE z INT;
  DECLARE CONTINUE HANDLER for SQLEXCEPTION (4)
    delete from t1 where c1 > z;
  BEGIN
    DECLARE CONTINUE HANDLER for SQLSTATE '54321' (2)
      BEGIN
        SET z = z + 1;
        RESIGNAL; (3)
      END;

    SELECT c1 INTO z FROM T1;
    IF(z > 100) THEN
      SIGNAL SQLSTATE '54321' SET MESSAGE_TEXT='custom error'; (1)
    END IF;
  END;
  SET z = 0; (5)
END;
```

DB2 Data Management Software



DB2 Data Management Software



@ business software

Flow Control Statements

IBM Software Group

Flow Control Statements

- CASE (selects an execution path (simple / searched))
- IF
- FOR (executes body for each row of table)
- WHILE
- ITERATE (forces next iteration. Similar to CONTINUE in C)
- LEAVE (leaves a block or loop. "Structured Goto")
- LOOP (infinite loop)
- REPEAT
- GOTO
- RETURN
- CALL (procedure call)

Simple CASE statement

```
CREATE PROCEDURE Update_Salary
(IN employee_num CHAR(6), IN rating INT)
LANGUAGE SQL
BEGIN
  CASE rating
    WHEN 1 THEN -- Note: WHEN argument is a value
      UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY *1.10,BONUS = 1000
        WHERE EMPNO = employee_num;
    WHEN 2 THEN
      UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY *1.05,BONUS = 500
        WHERE EMPNO = employee_num;
    ELSE
      UPDATE CORPDATA.EMPLOYEE
        SET SALARY = SALARY *1.03,BONUS = 0
        WHERE EMPNO = employee_num;
  END CASE;
END
```

Searched CASE statement

```
CREATE PROCEDURE Update_Salary
(IN employee_num CHAR(6), IN rating INT)
LANGUAGE SQL
BEGIN
  CASE -- Note: WHEN argument is a condition
    WHEN rating >= 1 AND rating < 4 THEN
      UPDATE CORPDATA.EMPLOYEE
      SET SALARY = SALARY *1.10,BONUS = 1000
      WHERE EMPNO = employee_num;
    WHEN rating >= 4 AND rating < 8 THEN
      UPDATE CORPDATA.EMPLOYEE
      SET SALARY = SALARY *1.05,BONUS = 500
      WHERE EMPNO = employee_num;
    ELSE
      UPDATE CORPDATA.EMPLOYEE
      SET SALARY = SALARY *1.03,BONUS = 0
      WHERE EMPNO = employee_num;
  END CASE;
END
```

IF statement

```
CREATE PROCEDURE Update_Salary
(IN employee_num CHAR(6), IN rating INT)
LANGUAGE SQL
BEGIN
  IF rating = 1 THEN
    UPDATE CORPDATA.EMPLOYEE
    SET SALARY = SALARY *1.10,BONUS = 1000
    WHERE EMPNO = employee_num;
  ELSEIF rating = 2 THEN
    UPDATE CORPDATA.EMPLOYEE
    SET SALARY = SALARY *1.05,BONUS = 500
    WHERE EMPNO = employee_num;
  ELSE
    UPDATE CORPDATA.EMPLOYEE
    SET SALARY = SALARY *1.03,BONUS = 0
    WHERE EMPNO = employee_num;
  END IF;
END
```

QuickLab: Conditional Statements (5 mins)

- Create a procedure that meets the following criteria

- ▶ Procedure Name = COND_TEST
- ▶ Output Parameter = v_result INT
- ▶ Requirement:

DB2 has a table called sysibm.sysdummy1 with 1 row and 1 column (IBMREQD). Determine the value of this row.

If the value is 'N' then set v_return to 0. If the value is 'Y' then set v_return to 1. For all other cases, return -1

QuickLab Solution : IF Statement

```
CREATE PROCEDURE IF_TEST (OUT v_result INT)
LANGUAGE SQL
BEGIN
    DECLARE v_tmp VARCHAR(1);
    SELECT IBMREQD INTO v_tmp FROM sysibm.sysdummy1;

    IF (v_tmp = 'Y') THEN
        SET v_result = 1;
    ELSEIF (v_tmp = 'N') THEN
        SET v_result=0;
    ELSE
        SET v_result=-1;
    END IF;
END
```

QuickLab Solution: slightly better solution

```
CREATE PROCEDURE IF_TEST (OUT v_result INT)
LANGUAGE SQL
BEGIN
    SET v_result = (SELECT CASE (IBMREQD)
                        WHEN 'Y' THEN 1
                        WHEN 'N' THEN 0
                        ELSE -1
                    END CASE
                FROM sysibm.sysdummy1);
END
```

FOR statement

- FOR Loop is the same as a READ ONLY cursor and is the most **convenient** way to iterate over a result set. (Searched UPDATE/DELETE still supported.)

```
CREATE PROCEDURE Concat_names()
LANGUAGE SQL
BEGIN -- Note: implicit cursor manipulation
    DECLARE fullname CHAR(140);
    FOR v1 AS SELECT firstnme, midinit, lastname
        FROM employee
    DO
        SET fullname = v1.lastname || ', ' || v1.firstnme ||
            ' ' || v1.midinit;
        INSERT INTO tname VALUES (fullname);
    END FOR;
END
```


REPEAT statement

- Use REPEAT when something must be done at least once

```
CREATE PROCEDURE Concat_names()
LANGUAGE SQL
BEGIN
    DECLARE at_end INT DEFAULT 0;
    DECLARE fullname CHAR(140);
    DECLARE first, mid, last VARCHAR(50);
    DECLARE cur1 CURSOR FOR SELECT first, mid, last
        FROM employee;
    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET at_end = 1;

    OPEN cur1;
    REPEAT
        FETCH cur1 INTO first, mid, last;
        SET fullname = last || ' ' || first || ' ' || mid;
        INSERT INTO tname VALUES (fullname);
    UNTIL at_end <> 0
    END REPEAT;
```

END

DB2 Data Management Software



LOOP & WHILE

```
...
OPEN c1;
fetch_loop:
LOOP
    FETCH c1 INTO var1, var2, var3;
    IF condition THEN
        LEAVE fetch_loop;
    END IF;
END LOOP fetch_loop;
CLOSE c1;
...
```

LOOP

```
...
OPEN c1;
WHILE ctr < (rows / 2 + 1) DO
    FETCH c1 INTO medianSalary;
    SET ctr = ctr + 1;
END WHILE;
CLOSE c1;
...
```

WHILE

DB2 Data Management Software



GOTO, LEAVE, ITERATE

```
lab: BEGIN
  DECLARE a INT DEFAULT 0;

  lab2: LOOP
    IF a > 100 THEN
      LEAVE lab;
    ELSEIF a < 50 THEN
      ITERATE lab2;
    ELSE
      GOTO out;
    END IF;
  END LOOP;
END;
out: RETURN 0;
```

- LEAVE: exits blocks and loops
- ITERATE: forces new iteration
- GOTO: supercedes both, but less clear

Use of GOTO

- Use of GOTO is **considered poor programming style** and is discouraged. However, here are some considerations
 - The labeled statement and the GOTO statement must be in the same scope
 - If the GOTO statement is defined in a FOR statement, label must be defined inside the same FOR statement, excluding a nested FOR statement or nested compound statement
 - If the GOTO statement is defined in a compound statement, label must be defined inside the same compound statement, excluding a nested FOR statement or nested compound statement
 - If the GOTO statement is defined in a handler, label must be defined in the same handler, following the other scope rules
 - If the GOTO statement is defined outside of a handler, label must not be defined within a handler.

Nested Stored Procedure CALLs

- an SQL calling procedure may call any procedure written in SQL or C, but not procedures written in other languages
- limited 16 levels of nesting
- can be recursive (up to 16)
 - ▶ limitation: must close all open cursors before calling itself

Nested CALL statement, with cursors

- When a calling SQL procedure expects to receive a result set from a target SQL procedure, **ALLOCATE CURSOR** and **ASSOCIATE RESULT SET LOCATOR** statements are used

```
CREATE PROCEDURE Caller(INOUT total INT)
LANGUAGE SQL
BEGIN
  DECLARE a, sqlcode INT DEFAULT 0;
  DECLARE LOC1 RESULT_SET_LOCATOR VARYING;

  SET total = 0;
  CALL Callee(a);

  ASSOCIATE RESULT SET LOCATOR(LOC1) WITH
    PROCEDURE Callee;
  ALLOCATE C1 CURSOR FOR RESULT SET LOC1;

  FETCH FROM C1 INTO a;
  WHILE sqlcode = 0 DO
    SET total = total + a;
    FETCH FROM C1 INTO a;
  END WHILE;
END
```

```
CREATE PROCEDURE Callee(IN p1 INT)
LANGUAGE SQL
RESULT SETS 1
BEGIN
  DECLARE cur1 CURSOR WITH RETURN TO CALLER FOR
    SELECT C1 FROM TABLE1 WHERE C1>p1;

  OPEN cur1;
END
```

Notes

Receiving Result Sets as a Caller

When you expect your calling SQL procedure to receive a result set from a target SQL procedure, you must use the `ALLOCATE CURSOR` and `ASSOCIATE RESULT SET LOCATOR` statements to access and use the result set.

ASSOCIATE RESULT SET LOCATOR

After a `CALL` statement to a target SQL procedure that returns one or more result sets to the caller, your calling SQL procedure should issue this statement to assign result set locator variables for each of the returned result sets. For example, a calling SQL procedure that expects to receive three result sets from a target SQL procedure could contain the following SQL:

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;
CALL targetProcedure();
ASSOCIATE RESULT SET LOCATORS(result1,result2,result3)
WITH PROCEDURE targetProcedure;
```

Notes

ALLOCATE CURSOR

Use the `ALLOCATE CURSOR` statement in a calling SQL procedure to open a result set returned from a target SQL procedure. To use the `ALLOCATE CURSOR` statement, the result set must already be associated with a result set locator through the `ASSOCIATE RESULT SET LOCATORS` statement. Once the SQL procedure issues an `ALLOCATE CURSOR` statement, you can fetch rows from the result set using the cursor name declared in the `ALLOCATE CURSOR` statement. To extend the previously described `ASSOCIATE LOCATORS` example, the SQL procedure could fetch rows from the first of the returned result sets using the following SQL:

```
DECLARE result1 RESULT_SET_LOCATOR VARYING;
DECLARE result2 RESULT_SET_LOCATOR VARYING;
DECLARE result3 RESULT_SET_LOCATOR VARYING;
CALL targetProcedure();
ASSOCIATE RESULT SET LOCATORS(result1,result2,result3)
WITH PROCEDURE targetProcedure;
ALLOCATE rsCur CURSOR FOR result1;
WHILE (at_end =0)DO
SET total1 =total1 +var1;
SET total2 =total2 +var2;
FETCH FROM rsCur INTO var1,var2;
ENDWHILE;
```

GET DIAGNOSTICS (RETURN_STATUS)

```
CREATE PROCEDURE Caller(p1 INT)
LANGUAGE SQL
BEGIN
  DECLARE a INT DEFAULT 0;

  CALL Callee(p1);
  GET DIAGNOSTICS a = RETURN_STATUS;
END
```

```
CREATE PROCEDURE Callee(IN p1 INT)
LANGUAGE SQL
BEGIN
  RETURN p1 + 100;
END
```

Notes:

The parameter bound for the return value parameter marker in the escape clause CALL syntax (?=CALL...) in a CLI/Java application

RETURN_STATUS identifies the status value returned from the stored procedure associated with the previously executed SQL statement, provided that the statement was a CALL statement. **If the previous statement is not such a statement, the value returned has no meaning and can be any integer.**

GET DIAGNOSTICS (ROW_COUNT)

- Indicates how many rows were affected in previous query or an **estimate** of how many rows will be in a result set if issued after PREPARE

```
CREATE PROCEDURE howmany
  ( IN deptnbr VARCHAR (3), OUT num INTEGER)
BEGIN

  DECLARE rcount INTEGER ;
  UPDATE PROJECT
    SET PRSTAFF = PRSTAFF +1.5
    WHERE DEPTNO = deptnbr;
  GET DIAGNOSTICS rcount = ROW_COUNT;
  SET num = rcout;
END
```

GET DIAGNOSTICS (EXCEPTION 1)

- Retrieves the DB2 error or warning information related to the SQL statement just executed.
- MESSAGE_TEXT retrieves the actual message
- DB2_TOKEN_STRING retrieves only the error/warning tokens

Quicklab: GET DIAGNOSTICS

- Create the following table from the CLP:

```
CREATE TABLE t1 (c1 int not null primary key);
```
- Then, create the following procedure

```
CREATE PROCEDURE insert_with_diag  
(IN a INT, IN b INT, OUT msg VARCHAR(250))  
SPECIFIC insert_with_diag  
BEGIN  
    DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING  
    BEGIN  
        GET DIAGNOSTICS EXCEPTION 1 msg = MESSAGE_TEXT;  
        ROLLBACK;  
    END;  
  
    IF a > b THEN SIGNAL SQLSTATE '80000'  
        SET MESSAGE_TEXT = 'A cannot be smaller than B';  
    END IF;  
  
    INSERT INTO t1 values (a);  
    INSERT INTO t1 values (b);  
END
```

Quicklab (cont...)

- Try the following test cases with insert_with_diag:

CALL insert_with_diag (1,2,?)

CALL insert_with_diag (1,1,?)

CALL insert_with_diag (2,1,?)

Dynamic SQL

Dynamic SQL

- Useful when the final form of SQL is known only at RUN TIME
- example: if col1 and tabname are variables:

```
'SELECT ' || col1 || ' FROM ' || tabname;
```
- Also recommended for DDL to avoid dependency problems and package invalidation
- Keywords:
 - ▶ EXECUTE IMMEDIATE - ideal for single execution SQL
 - ▶ PREPARE + EXECUTE - ideal for multiple execution SQL
- dynamic CALL statement supported using EXECUTE [USING ...] [INTO ...]
 - ▶ not supported in V7.x

Quicklab: Dynamic SQL

```
prerequisite: create table T2 (c1 int, c2 int)
```

```
CREATE PROCEDURE dyn1 (IN value1 INT, IN value2 INT)
```

```
SPECIFIC dyn1
```

```
BEGIN
```

```
  DECLARE stmt varchar(255);
```

```
  DECLARE st STATEMENT;
```

```
  SET stmt = 'insert into T2 values (?, ?)';
```

```
  PREPARE st FROM stmt;
```

```
  EXECUTE st USING value1, value1;
```

```
  EXECUTE st USING value2, value2;
```

```
  SET stmt = 'insert into T2 values (9,9)';
```

```
  EXECUTE IMMEDIATE stmt;
```

```
END
```

Compile stmt once

Execute it many times

Compile and execute once

Quicklab: Dynamic Cursors

```
CREATE PROCEDURE dyn2 (IN value1 INT)
```

```
LANGUAGE SQL
```

```
BEGIN
```

```
  DECLARE stmt varchar(255);
```

```
  DECLARE st STATEMENT;
```

```
  DECLARE cur1 CURSOR WITH RETURN FOR st; ☐ Dynamic cursor
```

```
  SET stmt = 'select * from t1 where c1 = ?';
```

```
  PREPARE st FROM stmt; ☐ Compile statement
```

```
  OPEN cur1 USING value1; ☐ Return result set
```

```
END
```

DB2 Data Management Software

IBM

DB2 Data Management Software

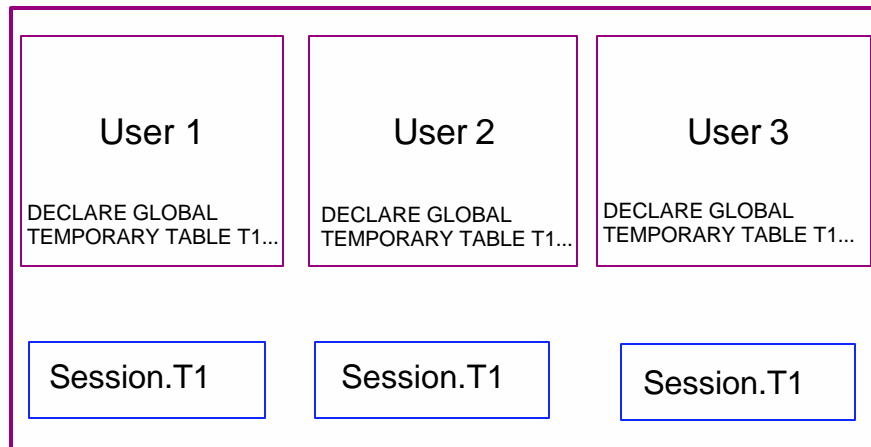
IBM

@ business software

Related Features

IBM Software Group

Declared Temporary Tables' Scope



- SESSION qualifier is required
- Definer of table granted all table privileges

Notes

A declared temporary table is a temporary table that is only accessible to SQL statements that are issued by the application which created the temporary table. A declared temporary table does not persist beyond the duration of the connection of the application to the database.

Before you can create a temporary table, ensure that a `USER TEMPORARY TABLESPACE` exists. No user temporary table spaces exist when a database is created. At least one user temporary table space should be created with appropriate `USE` privileges to allow definition of declared temporary tables.

The schema for declared temporary tables is always `SESSION`. To use the declared temporary table in your SQL statements, you must refer to the table using the `SESSION` schema qualifier.

To avoid confusing persistent tables with declared temporary tables, you should not create persistent tables using the `SESSION` schema. DB2 always resolves references to persistent and declared temporary tables with identical qualified names to the declared temporary table.

Declared temporary tables are subject to a number of restrictions. For example, you cannot define aliases, or views for declared temporary tables. You cannot use `IMPORT` and `LOAD` to populate declared temporary tables.

In version 7, you cannot create indexes on declared temporary tables. (Restriction has been lifted in Version 8)

Restrictions on Temp Tables

- Cannot be specified on an ALTER, COMMENT, GRANT, LOCK, RENAME, or REVOKE
- Cannot be referenced in a CREATE ALIAS, CREATE FUNCTION, CREATE TRIGGER, or CREATE VIEW
- Cannot be specified in referential constraints
- Cannot use IMPORT or LOAD to populate
- Does not support
 - ▶ LOB-type columns
 - ▶ User-defined type columns
 - ▶ LONG VARCHAR columns
 - ▶ DATALINK columns
- Must have User Temporary Table Space defined

End of Transaction Impact

- Commit
 - ▶ by default,
 - if no WITH HOLD cursor, all rows deleted
 - if WITH HOLD cursor, rows not deleted
 - ▶ option: ON COMMIT PRESERVE ROWS
- Rollback
 - ▶ if session included DECLARE GLOBAL TEMPORARY TABLE T1, table is dropped
 - ▶ if session included DROP TABLE SESSION.T1, table is recreated with no rows
 - ▶ if session modified data in SESSION.T1, all rows are deleted

QuickLab: Temporary Tables

Perform the following using the CLP

```
CONNECT to sample

DECLARE GLOBAL TEMPORARY TABLE T1 (EMPNO CHAR(10), SALARY
int) ON COMMIT PRESERVE ROWS NOT LOGGED

INSERT INTO SESSION.T1 (SELECT empno, salary FROM EMPLOYEE
WHERE salary > 40000)

SELECT empno FROM SESSION.T1 order by salary

COMMIT

SELECT empno FROM SESSION.T1 order by salary

CONNECT RESET

CONNECT to sample

SELECT empno FROM SESSION.T1 order by salary
==> ERROR! (SQL0204)
```

Application Savepoints

```
SAVEPOINT S1 ON ROLLBACK RETAIN CURSORS;
INSERT INTO T2 (C1, C2) VALUES (2,3);
INSERT INTO T2 (C1, C2) VALUES (2,1);
INSERT INTO T2 (C1, C2) VALUES (245,5);
ROLLBACK TO SAVEPOINT S1;
RELEASE SAVEPOINT S1;
```

```
SAVEPOINT S1 ON ROLLBACK RETAIN CURSORS;
INSERT INTO T2 (C1, C2) VALUES (2,300);
INSERT INTO T2 (C1, C2) VALUES (200,3);
COMMIT;
```

Contents of T2:

C1	C2
2	300
200	3

Savepoint Considerations

- If savepoint contains DDL, ROLLBACK TO SAVEPOINT invalidates cursors dependent on DDL
- NOT LOGGED INITIALLY tables in savepoint treat ROLLBACK TO SAVEPOINT as ROLLBACK WORK
- If savepoint contains declared temporary tables, ROLLBACK TO SAVEPOINT drops temporary table
- SAVEPOINT, RELEASE SAVEPOINT, and ROLLBACK TO SAVEPOINT statements will flush buffers for buffered inserts
- ROLLBACK TO SAVEPOINT will not impact content of local copy of rows kept for blocked cursor; subsequent fetch may retrieve inserted rows which were rolled back
- RELEASE SAVEPOINT explicitly releases a savepoint. If not specified, savepoint it is released at the end of the transaction.

Savepoint Restrictions

- Cannot contain atomic compound SQL
- Cannot be issued in atomic compound SQL
- Cannot be nested
- Cannot be included in triggers or UDFs
- No limit to number of savepoints in a transaction
- SET INTEGRITY statements treated like DDL

Notes

You can issue a CLOSE statement to close invalid cursors. Other actions against an invalid cursor (such as FETCH, OPEN, UPDATE WHERE CURRENT OF, or DELETE WHERE CURRENT OF) will return negative SQLCODEs.

Within savepoints, DB2 treats tables with the NOT LOGGED INITIALLY property and temporary tables as follows:

- Within a savepoint, you can create a table with the NOT LOGGED INITIALLY property or alter a table to have the NOT LOGGED INITIALLY property. For these savepoints, however, DB2 treats ROLLBACK TO SAVEPOINT statements as ROLLBACK WORK statements and rolls back the entire transaction.
- If a temporary table is declared within a savepoint, a ROLLBACK TO SAVEPOINT statement drops the temporary table. If a temporary table is declared outside a savepoint, a ROLLBACK TO SAVEPOINT does not drop the temporary table.

If your application takes advantage of both buffered inserts and savepoints, DB2 flushes the buffer before executing SAVEPOINT, RELEASE SAVEPOINT, or ROLLBACK TO SAVEPOINT statements.


If your application uses savepoints, consider preventing cursor blocking by precompiling or binding the application with the precompile option BLOCKING NO. While blocking cursors can improve the performance of your application by pre-fetching multiple rows, the data returned by an application that uses savepoints and blocking cursors may not reflect data that has been committed to the database.

If you precompile the application using blocking, and your application issues a FETCH statement after a ROLLBACK TO SAVEPOINT has occurred, the FETCH statement may retrieve rows which were inserted during the savepoint and subsequently rolled back.

If there are any active savepoints in an application when an XA compliant transaction manager issues an XA_END request, DB2 issues a RELEASE SAVEPOINT statement.

DB2 Data Management Software

IBM

 e-business software

Coding for Performance

IBM Software Group

Coding tips for Performance

- **IMPORTANT:** If there is a choice between using procedural logic or SQL, use SQL

- ▶ Example:

```
CREATE PROCEDURE fire_employee(IN mid CHAR(1))
...
  DECLARE cur1 CURSOR FOR SELECT midinit FROM employee;

  OPEN cur1;
  REPEAT
    FETCH cur1 INTO middle;
    IF (middle = mid AND SQLCODE = 0) THEN
      DELETE FROM employee WHERE CURRENT OF cur1;
    END IF;
  UNTIL SQLCODE <> 0
  END REPEAT;
END
```

- ▶ Versus:

```
CREATE PROCEDURE fire_employee(IN mid CHAR(1))
...
  DELETE FROM EMPLOYEE WHERE midinit = mid;
END
```

- ▶ The second option is *much* faster

Coding tips for Performance

- Avoid using DROP statements
 - ▶ Package associated with stored procedure will be invalidated and must be dynamically rebuilt upon next execution of the stored procedure
- To assign values to a set number of variables
 - ▶ use VALUES INTO clause rather than multiple SET statements:
 - SET var1 = 100; SET var2 = 200; SET var3 = 200+1; OR
 - VALUES(100, 200, 200+1) INTO var1,var2,var3;
 - ▶ VALUES method can be performed in parallel
 - ▶ Each set statement must check for SQLExceptions, for example - Overflow. This can be quite costly.

Deploying Stored Procedures

IBM Software Group

Moving Stored Procedures

- **GET ROUTINE INTO <filename> FROM [SPECIFIC]
PROCEDURE <procname> [hide body]**

where:

<filename> : filename or path/filename

<procname>: (specific) proc name. If
unqualified, the default is
the current schema.

hide your
source code
on target
server

Moving Stored Procedures

- **PUT ROUTINE FROM <filename>**
- **PUT ROUTINE FROM <filename> OWNER <newowner>**
- **PUT ROUTINE USING <filename> OWNER <newowner> USE REGISTERS**

- Inputs
 - ▶ filename: filename or path/filename
 - ▶ newowner: authorization-name for routine
 - default: authorization-name of original definer of routine

Note: Compiler not required on target server. Target server must be same hardware, database and OS version

Recommended Book



More info:
<http://www.software.ibm.com/data/developer/sqlplbook>

DB2 SQL Procedural Language for Linux, UNIX,
and Windows

ISBN: 0-13-100772-6