



IBM Software Group

Native SQL Procedures in DB2 9 for z/OS

Fen-Ling Lin
Senior Technical Member Stuff and Manager
Query Technology, DB2 for z/OS
IBM Silicon Valley Laboratory



DB2 for z/OS Technical Conference
October 5-6, 2009
Taipei, Taiwan

 ON DEMAND BUSINESS™

©2008 IBM Corporation

Agenda

- Introduction
 - Background and Motivation
 - Comparing external vs native SQL procedures
- New Features
 - Enhancements
 - Versioning
 - Deployment
 - Debugging
- Performance Technologies
- Best Use of Storage



What is an SQL procedure ?

- A stored procedure that contains only SQL statements.
- May use SQL control statements to write the logic part of the program (WHILE, IF, etc)
- **SQL Procedural Language** or **SQL PL**



External and Native SQL procedures

- **External** SQL procedures (from V5 on)
 - Generated C program which runs in a WLM environment
- **Native** SQL procedures (from V9)
 - The SQL procedure logic runs in the DBM1 address space



An example of SQL procedure

```
BEGIN
  DECLARE v_numRecords INTEGER DEFAULT 1;
  DECLARE v_counter INTEGER DEFAULT 0;
  DECLARE c1 CURSOR FOR
    SELECT salary FROM staff ORDER BY salary;
  DECLARE c2 CURSOR WITH RETURN FOR
    SELECT name, job, salary
      FROM staff
      WHERE salary > medianSalary
      ORDER BY salary;
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET medianSalary = 0;
  SELECT COUNT(*) INTO v_numRecords FROM STAFF;
  OPEN c1;
  WHILE v_counter < (v_numRecords / 2 + 1) DO
    FETCH c1 INTO medianSalary;
    SET v_counter = v_counter + 1;
  END WHILE;
  CLOSE c1;
  OPEN c2;
```

```
CREATE PROCEDURE
MEDIAN_RESULT_SET
  (OUT medianSalary DECIMAL(7,2))
  DYNAMIC RESULT SETS 1
```

routine-body



An example of SQL procedure

```
BEGIN
  DECLARE v_numRecords INTEGER DEFAULT 1;
  DECLARE v_counter INTEGER DEFAULT 0;
  DECLARE c1 CURSOR FOR
    SELECT salary FROM staff ORDER BY salary;
  DECLARE c2 CURSOR WITH RETURN FOR
    SELECT name, job, salary
      FROM staff
      WHERE salary > medianSalary
      ORDER BY salary;
  DECLARE EXIT HANDLER FOR NOT FOUND
    SET medianSalary = 0;
  SELECT COUNT(*) INTO v_numRecords FROM STAFF;
  OPEN c1;
  WHILE v_counter < (v_numRecords / 2 + 1) DO
    FETCH c1 INTO medianSalary;
    SET v_counter = v_counter + 1;
  END WHILE;
  CLOSE c1;
  OPEN c2;
```

END

The condition handlers can be used to handle errors, warnings, not found, or other specified conditions.

Note that instead of host variables used in external procedures, SQL procedures use declared SQL variables and parameters which are used without “colon”s..



Values for the native SQL procedures

- Enhanced SQL PL support
 - Better Family Compatibility and Standards Compliance
 - Enhanced Portability
- Support for the Application Development Lifecycle
 - Support for the Versioning of procedures
 - Support for the Debugging of the procedures
 - Support for the Deployment of procedures
 - Security and the management of the source code
- Enhanced Performance
- Enhanced Usability
- Reduced cost of ownership



Enhanced SQL PL support and Portability

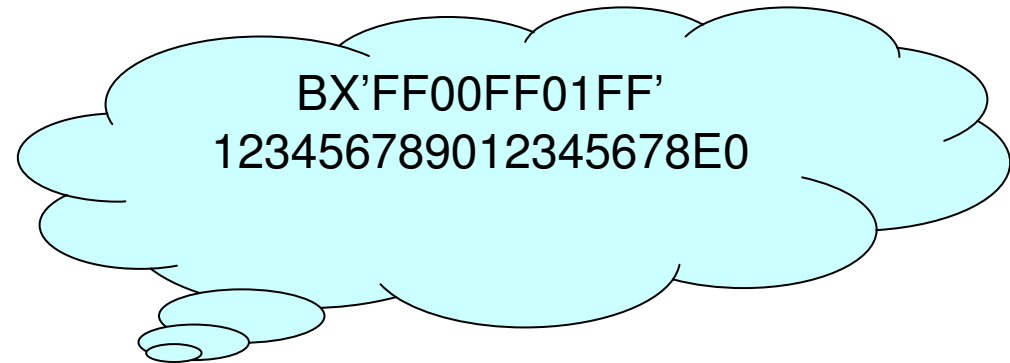
Native SQL procedures have enhanced support for the SQL Procedural Language, including the nested compound statements and more new data types. You can write complex SQL procedures with ease and the SQL procedures are more portable.

- Richer support for SQL PL
- Easier to program
- More portable
- More family compatible
- More compliance with the standards



More data types are supported

- BIGINT
- BINARY
- VARBINARY
- DECFLOAT

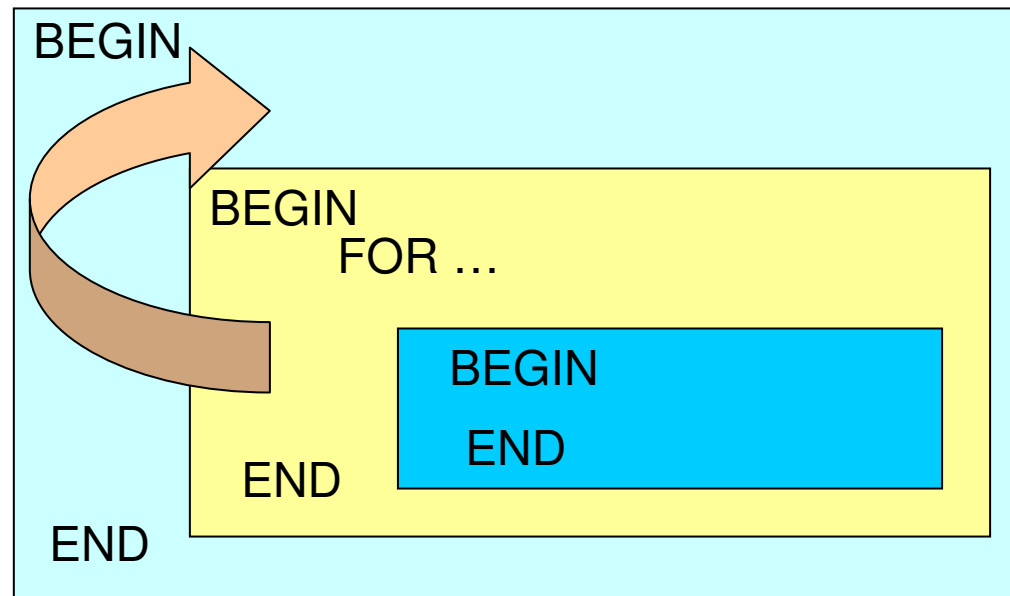


XML, UDTs, ROWIDs, LOB locators, LOB File reference are not supported.



More SQL PL constructs are supported

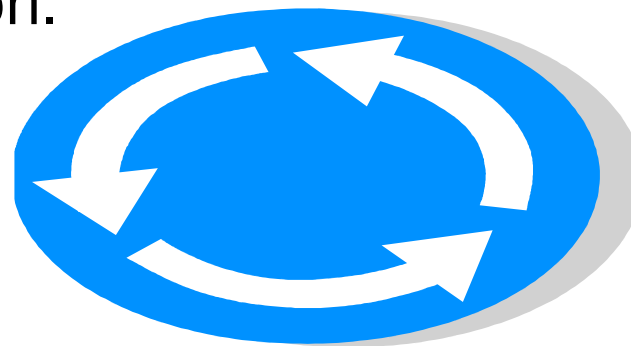
- Nested compound statements
- Multiple general conditions on a handler
- FOR loop
- Extended GOTO



Support for the Application Development Lifecycle

Native SQL procedures have been designed with the view of the application development life cycle in mind.

You can create a version of an SQL procedure, debug it, replace it or add a new version of the procedure, and finally deploy it into production.



Application life cycle enhancements in V9

- Extended versioning support (in the DB2 catalog)
- Unified Debugger support
- New syntax for CREATE PROCEDURE
- New syntax for ALTER PROCEDURE
- Deployment (new commands)

- Source code management
 - Security enhanced: source in catalog vs. external tables
 - Line feed and comments: aid in debugging
- Application and tools support (DSNTEP2, SPUFI)



Enhanced productivity, reduced cost of ownership, and more security

You will not need a C or C++ compiler to create native SQL procedures. The multiple steps of setup and level of complexity in the build process that are required by an external SQL procedure, has been simplified for a native SQL procedure.

DB2 manages the various aspects of the application development lifecycle in a consistent and integrated manner providing enhanced security, including the source code for the native SQL procedures.



Comparison of the external and native SQL procedures

- **Preparation**

- External: multi-step, require C compiler
- Native: single-step DDL

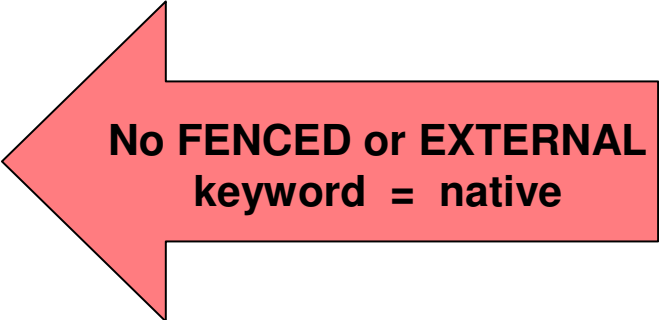
- **Execution**

- External: require WLM environment, load module
- Native: run entirely within the DB2 engine



SQL PL native procedure creation in V9 (NFM)

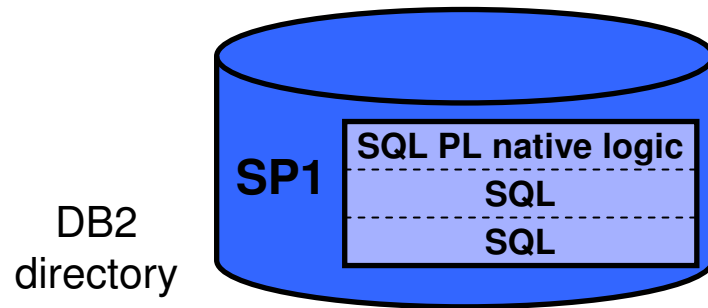
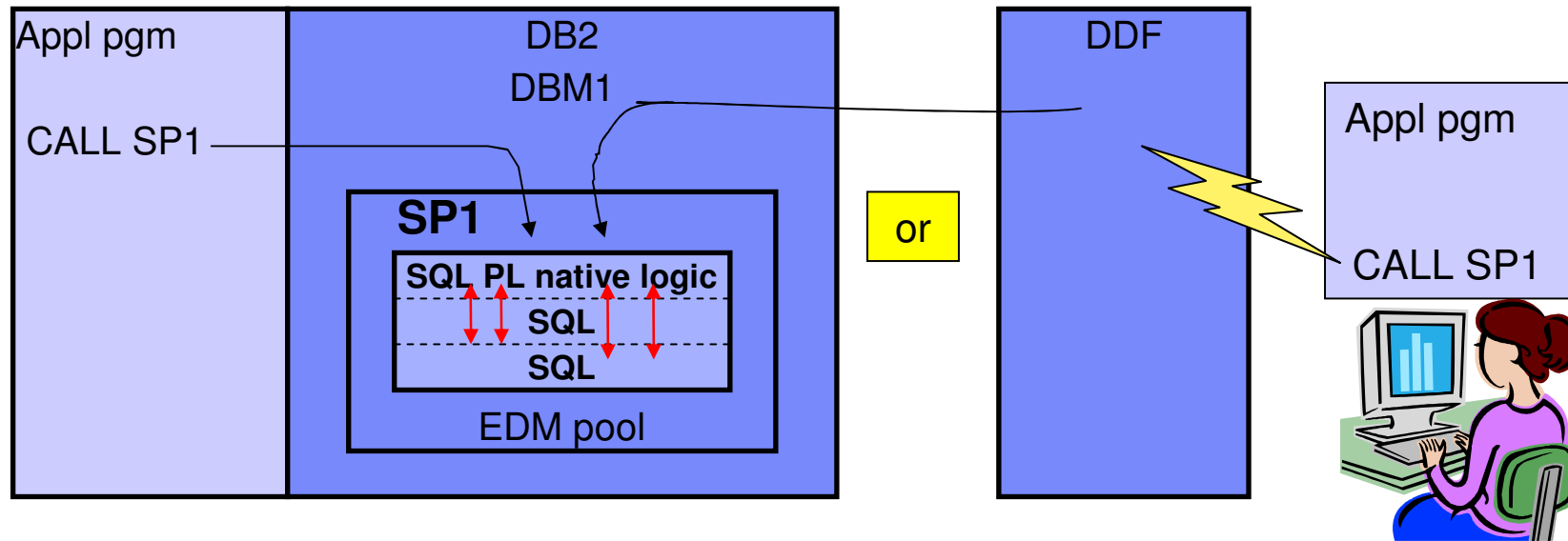
```
CREATE PROCEDURE MEDIAN_RESULT_SET
  (OUT medianSalary DECIMAL(7,2))
  ...
  DYNAMIC RESULT SETS 1
BEGIN
  DECLARE v_numRecords INTEGER DEFAULT 1;
  ...
  SELECT COUNT(*) INTO v_numRecords FROM staff;
  OPEN c1;
  WHILE v_counter < (v_numRecords/2+1)
    DO FETCH c1 INTO medianSalary;
    SET v_counter = v_counter + 1;
  END WHILE;
  CLOSE c1;
  ...
END
```



**No FENCED or EXTERNAL
keyword = native**

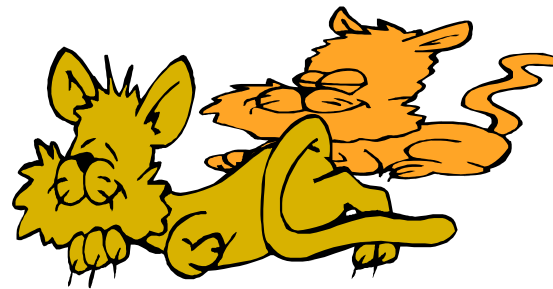


SQL PL native procedure execution in V9



SQL PL native procedure execution

- General performance improvement because of savings in API trips between the procedure application logic and the DBM1 address space → single trip for the entire routine
- “Active” version concept
- CURRENT ROUTINE VERSION special register



SQL PL native procedure versioning in V9

```
CREATE PROCEDURE MEDIAN_RESULT_SET
  (OUT medianSalary DECIMAL(7,2))
  VERSION MEDIAN_V1
  DYNAMIC RESULT SETS 1
BEGIN
  DECLARE v_numRecords INTEGER DEFAULT 1;
  ...
  SELECT COUNT(*) INTO v_numRecords FROM staff;
  OPEN c1;
  WHILE v_counter < (v_numRecords/2+1)
    DO FETCH c1 INTO medianSalary;
    SET v_counter = v_counter + 1;
  END WHILE;
  CLOSE c1;
  ...
END
```

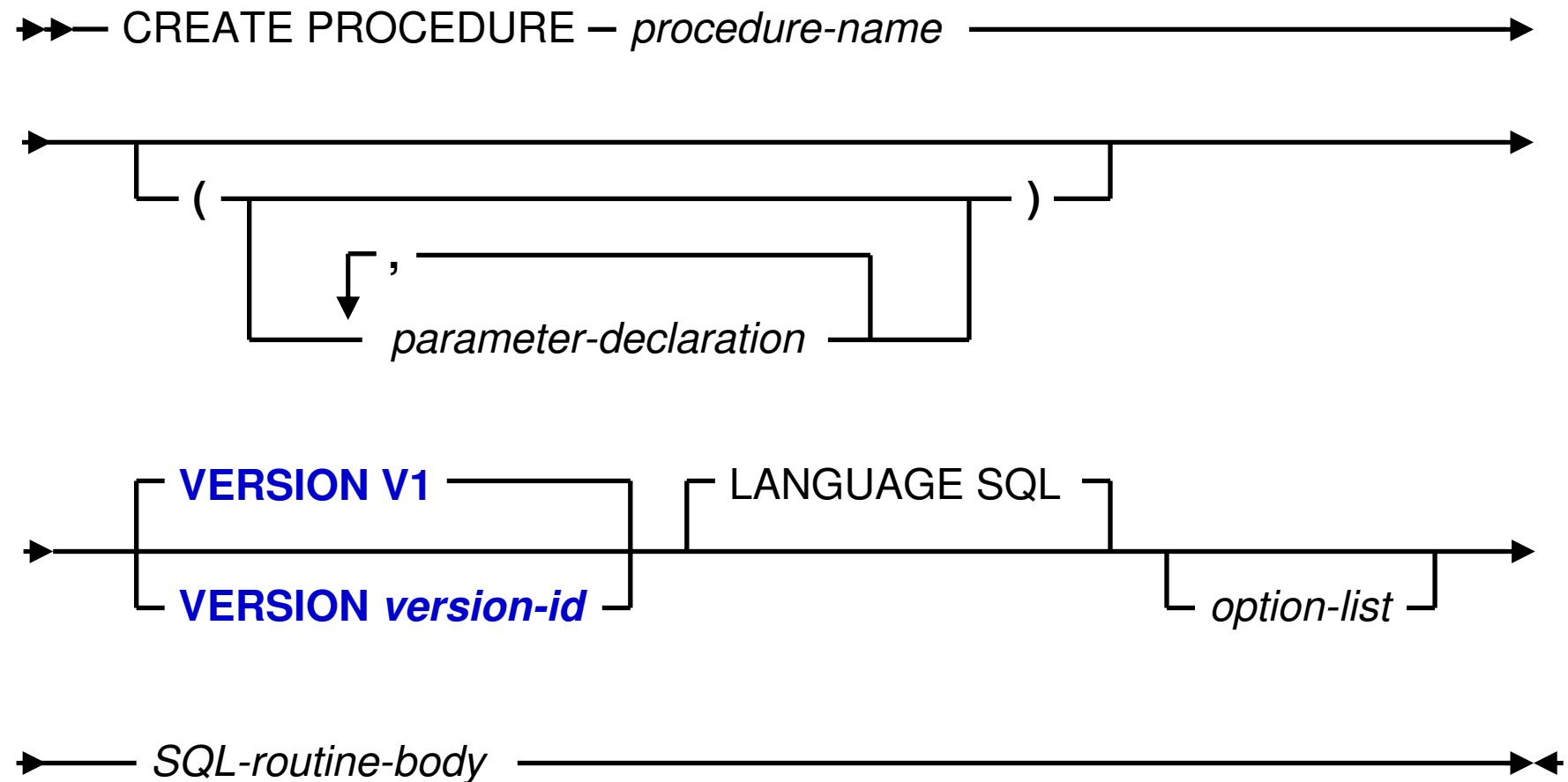


Versioning

- In V9, option **VERSION** is added in CREATE and ALTER statements for SQL PL procedures, so multiple versions can be created/added for the same stored procedure (with the same schema name).
- One of the versions is the **active** version.
- Any version of a stored procedure can be "promoted" to be the active version by ALTER.
- By default, the current active version will be the one to run when the stored procedure is called, unless CURRENT ROUTINE VERSION special register is set.



Creating a version of an SQL procedure in V9



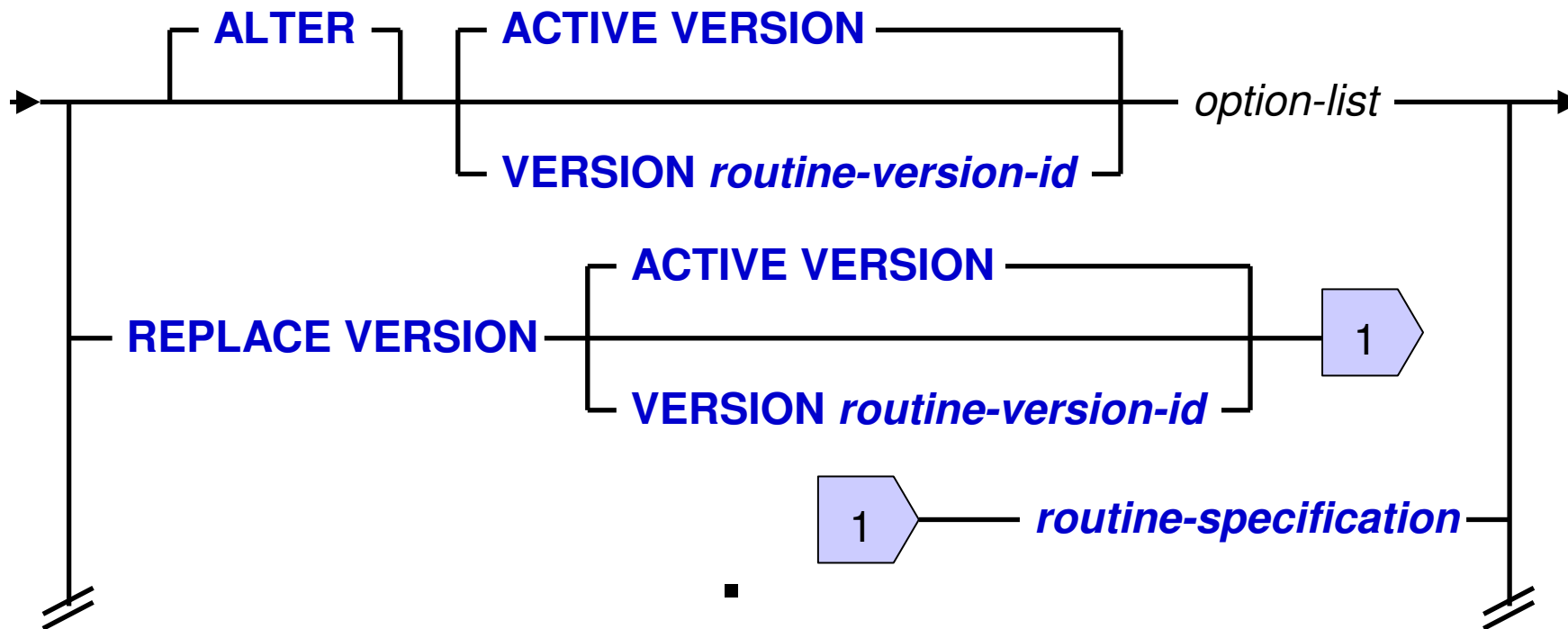
Altering V9 procedures

▶▶ ALTER PROCEDURE — *procedure-name* →

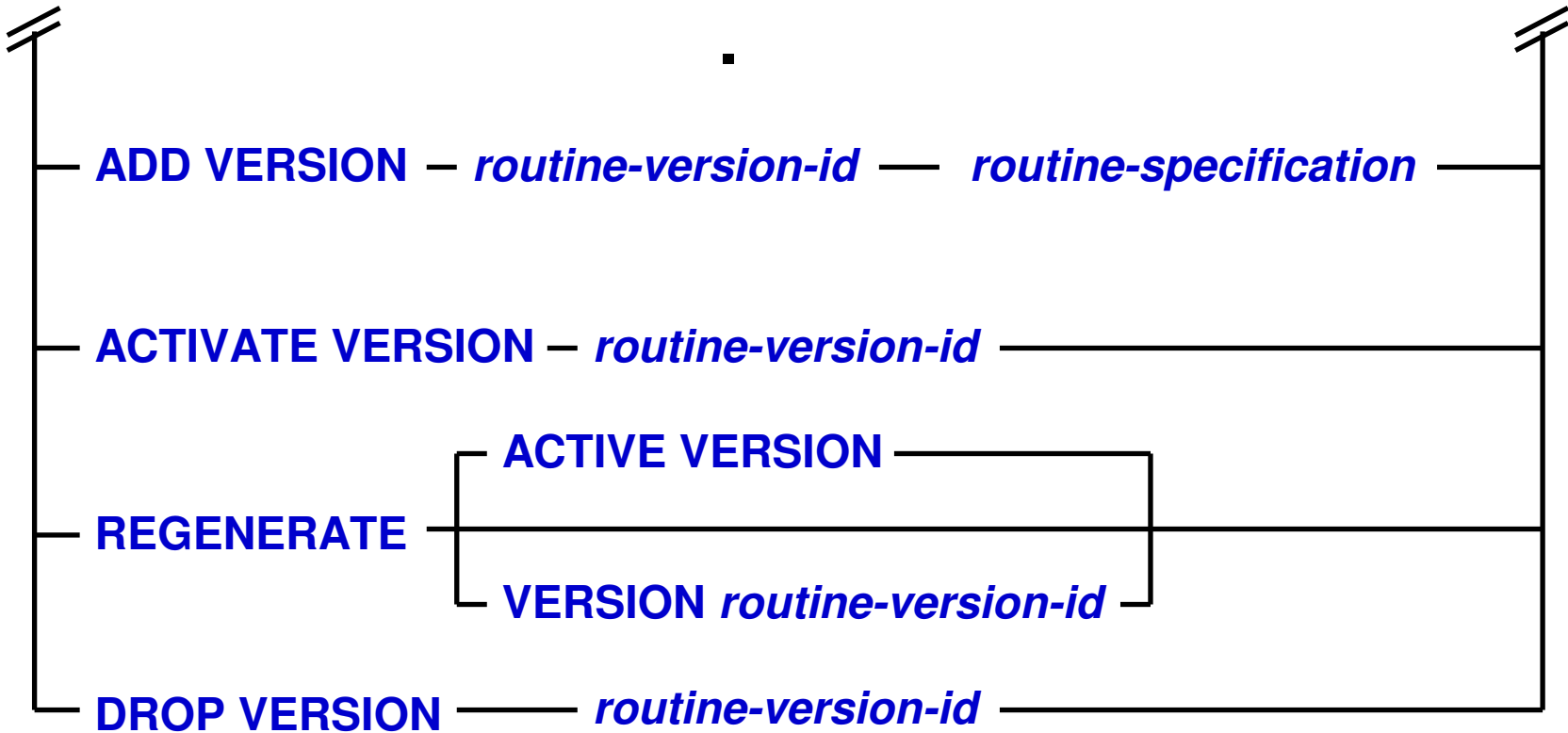
▶▶ *native-procedure-alteration*
external-procedure-alteration →



Native procedure alterations



Native procedure alterations . . . continued



ALTER examples

```
ALTER PROCEDURE UPDATE_SALARY  
  ALTER VERSION UPSALV1 ASUTIME 5000
```

```
ALTER PROCEDURE UPDATE_SALARY  
  ALTER ASUTIME 5000
```

```
ALTER PROCEDURE UPDATE_SALARY  
  REPLACE ACTIVE VERSION (IN EMPID CHAR(6),  
    IN RATE DECIMAL(7,2))  
MODIFIES SQL DATA  
UPDATE EMP SET SALARY=SALARY*RATE  
  WHERE EMPNO = EMPID
```

...



ALTER examples . . . continued

```
ALTER PROCEDURE UPDATE_SALARY
  ADD VERSION UPSALV2 (IN EMPID CHAR(6),
    IN RATE DECIMAL(7,2))
MODIFIES SQL DATA
UPDATE EMP SET SALARY=SALARY*RATE*(1.2)
  WHERE EMPNO = EMPID
```

...

```
ALTER PROCEDURE UPDATE_SALARY
  ACTIVATE VERSION UPSALV2
```

```
ALTER PROCEDURE UPDATE_SALARY
  REGENERATE ACTIVE VERSION
```



Calling a native SQL procedure

- Procedure resolution
 - schema (current path)
 - procedure name
 - number of parameters (no overloading for z/OS)

- Selecting a version to execute
 - CURRENT ROUTINE VERSION (special register)
 - e.g., useful for quick test after deployment
 - no catalog caching
 - Active VERSION (specified in the catalog)
 - default
 - catalog caching (same as before)



Impacts on other SQL statements

- COMMENT ON PROCEDURE statement
 - Extended to handle multiple versions of a procedure.
- GRANT and REVOKE statements
 - Privileges are the same for all versions of a procedure.
- DROP statement
 - Drop all versions of a procedure
 - To drop a version of a procedure, use ALTER PROCEDURE ... DROP VERSION ...
 - Extended to restrict the dropping of packages that implement a version of a procedure.



Impacts on SQL commands

- STOP / START PROCEDURE command
 - Affect all the versions of SQL procedures that will be stopped / started
- DISPLAY PROCEDURE command
 - Native SQL procedures are not reflected in the output
 - If a native SQL procedure is under the effect of a STOP PROCEDURE command, then the procedure name and status will be displayed, but the statistics will be 0



Upward compatibility

- External SQL procedures will continue to work in V9 either in CM or NFM
- External SQL procedures can continue to be created in V9 NFM
 - CREATE PROCEDURE ... FENCED or EXTERNAL keyword required
- Native SQL procedures can be created starting in V9 NFM
 - CREATE PROCEDURE ... (FENCED or EXTERNAL keyword not used)
- Both native and external SQL procedures can be called in V9 in NFM



FOR

- Executes one or multiple statements for each row of a table
- The cursor is defined with a SELECT statement which describes the rows and columns
- Statements within the FOR are executed for each row selected



FOR ... syntax

→ *label:* FOR *for-loop-name* AS →

→ *cursor-name* CURSOR *WITHOUT HOLD* FOR →
WITH HOLD

→ *select-statement* DO *SQL-procedure-statement* ; →

→ END FOR *label* →



FOR example

```
BEGIN
  DECLARE  fullname      CHAR(40);
  FOR      v1 AS
          c1 CURSOR FOR
          SELECT firstnme, midinit, lastname
          FROM   employee
          DO
            SET fullname =
              lastname CONCAT ', '
                  CONCAT firstnme
                  CONCAT ' '
                  CONCAT midinit ;
            INSERT INTO TNAMES VALUES ( fullname ) ;
  END FOR ;
END;
```



Name resolution -- external and native SQL PL

```
CREATE PROCEDURE . . .  
  BEGIN;  
    DECLARE dept CHAR(3);  
    DECLARE x CHAR(3);  
    :  
    DECLARE c1 CURSOR FOR  
      SELECT dept INTO x  
      FROM emp ;  
    :  
  END ;
```

- Ambiguity arises, since **dept** is both
 - declared as an SQL variable
 - a column in the table emp
- External SQL PL will match this **dept** to the SQL variable
- Native SQL PL, LUW, iSeries will match this **dept** to **emp.dept**



SQL PL -- better practice

```
CREATE PROCEDURE . . .
```

```
  STEP1 BEGIN;  
    DECLARE dept CHAR(3);  
    DECLARE x CHAR(3);  
    DECLARE y CHAR(3);  
    :  
    DECLARE c1 CURSOR FOR  
      SELECT STEP1.dept, emp.dept INTO x,y  
      FROM emp ;  
    :  
  END STEP1;
```

It is good practice to qualify names
if there is a potential for ambiguity



Compound SQL statements

- A compound statement contains a block of SQL statements and declarations for SQL variables, cursors, and condition handlers.
- In DB2 V8, the body of an SQL procedure could contain
 - a single compound statement (which could contain other SQL statements, except for another compound statement), or
 - a single SQL procedure statement other than the compound statement.
- Thus it was not possible to nest compound statements* within an SQL procedure. Additionally, this meant that a condition handler could not contain a compound statement.



DB2 V9 supports for nested compound statements

- With the support for nested compound statements for native SQL procedures:
 - A compound statement can now be used within a condition handler.
 - Nested compound statements can be used to define different scopes for SQL variables, cursors, condition names, and condition handlers.



Compound within condition handlers

- You can now use a compound statement within the declaration of a condition handler

```
BEGIN
  DECLARE SQLSTATE CHAR(5);
  DECLARE PrvSQLState CHAR(5) DEFAULT '00000';
  DECLARE ExceptState INT;
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN
      SET PrvSQLState = ...
      SET ExceptState = ...
      ...
    END;
END
```

PK43524
Stacked
Diagnostics
Area



Don't use
IF (1=1) THEN
...
...
END IF;

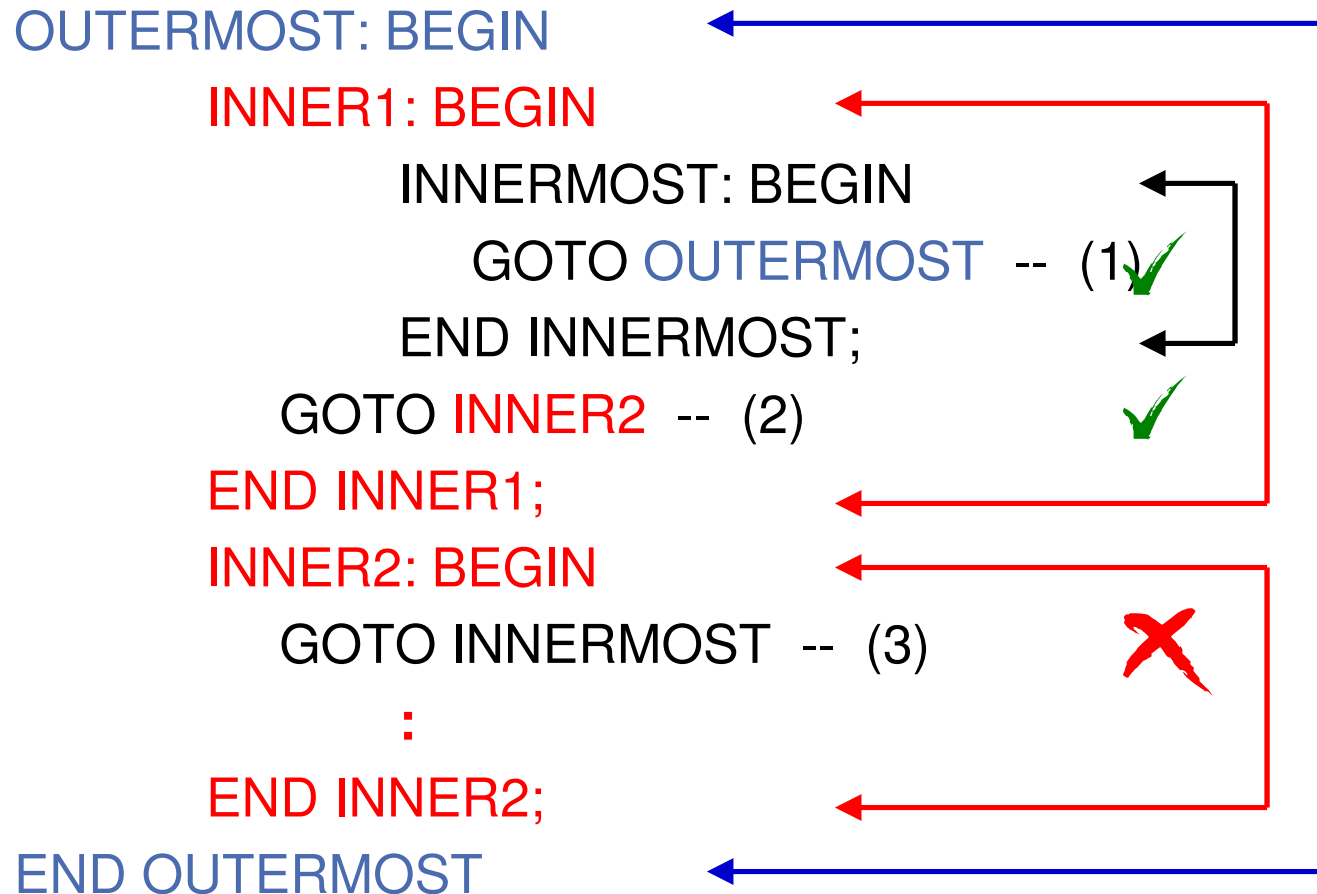


Using labels to define scope

- Nested compound statements can be used within an SQL procedure to define the scope of
 - SQL variable declarations
 - cursors
 - condition names
 - and condition handlers
- Each compound statement has its own defined scope, and can have a **label**.

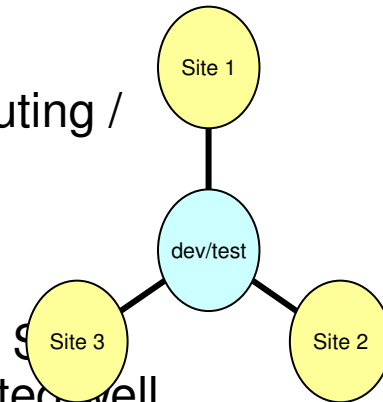


Enhanced GOTO support



Deployment

- Deployment of an SQL PL procedure is the step of distributing / installing the procedure created on one system to other system(s).
- Deployment is useful for customers who want to install an SQL PL procedure to a production system after it has been tested well.
- Deployment is different from remote BIND package, because the logic of the procedure body (stored as a special section in the package) will not be re-bound. Customers do not need to worry about unexpected behavior change after the deployment.



Deployment prior to V9 new feature

- Prior to V9, customers deploy SQL stored procedures by
 - Copying over the load modules of the stored procedures (this ensures that the logic of the stored procedure body is not changed after deployment)
 - Sending DBRM for the stored procedure over and issuing a BIND PACKAGE
 - Issuing CREATE PROCEDURE to define the procedure
- Keeping the stored procedure body logic intact is critical because customers need a smooth move from a testing environment to a production environment



Deployment as V9 new feature

- Deployment of SQL PL procedures in V9 is done via a new BIND PACKAGE option: **DEPLOY**
- Example: after the following CREATE PROCEDURE statement, which creates procedure TEST.MYPROC at the current site (for testing), the BIND PACKAGE command with DEPLOY option deploys the stored procedure onto a remote production system.

```
CREATE PROCEDURE TEST.MYPROC VERSION V1 ...  
  BEGIN  
  ...  
  END
```

```
BIND PACKAGE(CHICAGO.PRODUCTION) DEPLOY (TEST.MYPROC)  
COPYVER(V1) ACTION(ADD) QUALIFIER(XYZ)
```



Enhanced Performance

Native SQL procedures will be executed entirely in the DB2 engine, whereas external SQL procedures are executed in the WLM environment.

The native SQL procedures are expected to outperform typical external SQL procedures.



Performance technologies



- Execution within DB2 engine
- SQLPL Compiler Transformation technology
- Global dynamic virtual storage technology
- zIIP enabled for offloading



Best use of storage

- Above the bar storage is utilized
- LOBs handling via locators
- Reuse of storage by overlapping
- Global dynamic virtual storage



Summary

- Native SQL procedures have enhanced support for the SQL Procedural Language, including the nested compound statements and more new data types. You can write complex SQL procedures with ease and the SQL procedures are more portable.
- Native SQL procedures have been designed with the view of the application development lifecycle in mind. You can create a version of an SQL procedure, debug it, replace it or add a new version of the procedure, and finally deploy it into production.
- DB2 manages the various aspects of the application development lifecycle in a consistent and integrated manner providing enhanced security, including the source code for the native SQL procedures.
- Native SQL procedures will be executed entirely in the DB2 engine, whereas external SQL procedures are executed in the WLM environment. The native SQL procedures are expected to outperform typical external SQL procedures.
- You will not need a C or C++ compiler to create native SQL procedures. The multiple steps of setup and level of complexity in the build process that are required by an external SQL procedure, has been simplified for a native SQL procedure.

