How to Write SQL Stored Procedures

Rob Bestgen
Db2 for i Consultant
bestgen@us.ibm.com

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
  - Silly quiz: what does PSM stand for?

- 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

SQL Procedures

- A **procedure** is SQL’s version of a program
- Similar to any other high-level language program
- Invoked with a CALL statement
- Supports parameters, both input and output
  – and Result Sets!
- Allows natural interaction between logic programming and database access in a single (SQL) language

Procedure

1 Create it (one time, like a pgm)

```
CREATE OR REPLACE PROCEDURE total_val (IN Member# CHAR(6),
                                       OUT total DECIMAL(12,2))
  LANGUAGE SQL
    BEGIN
       CALL refresh_accounts;
       SELECT SUM(curr_balance) INTO total
         FROM accounts
        WHERE account_owner=Member# AND
            account_type IN ('C','S','M');
    END
```

2 CALL it (many times) from an SQL interface

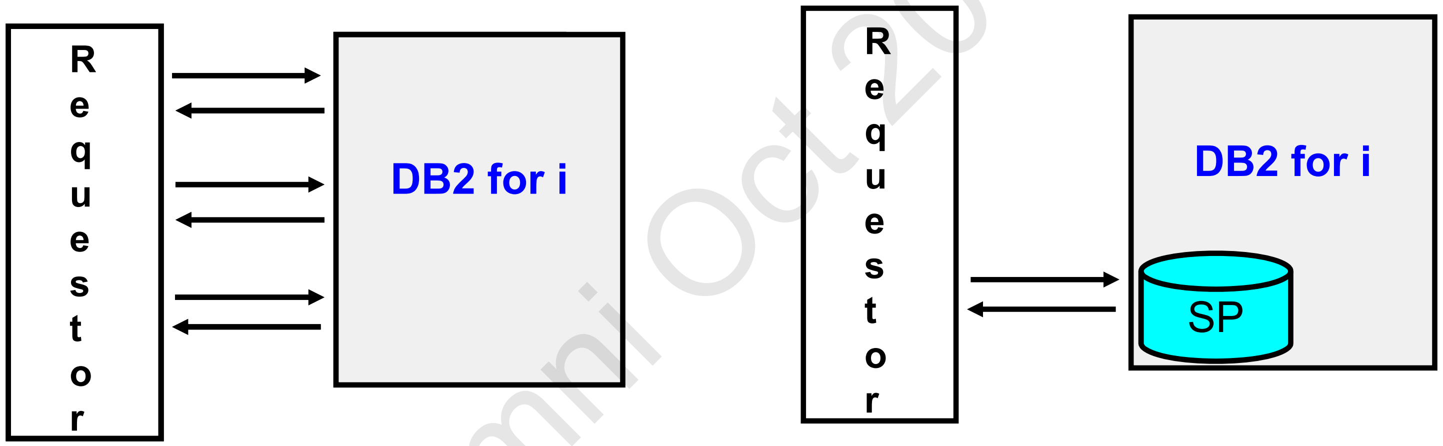
```
CALL total_val('135790', balance_var)
```

Why use SQL procedures?

- Combines application logic with data access
  - One language, natural interaction
- Encapsulate complex logic
- Can leverage security control using owner adoption
- Provide underpinning behind service interfaces (like REST, microservices)
- Reduce data movement (network traffic)
- Enables (batch) programmatic processing of SQL
- Error handling/recovery for data access

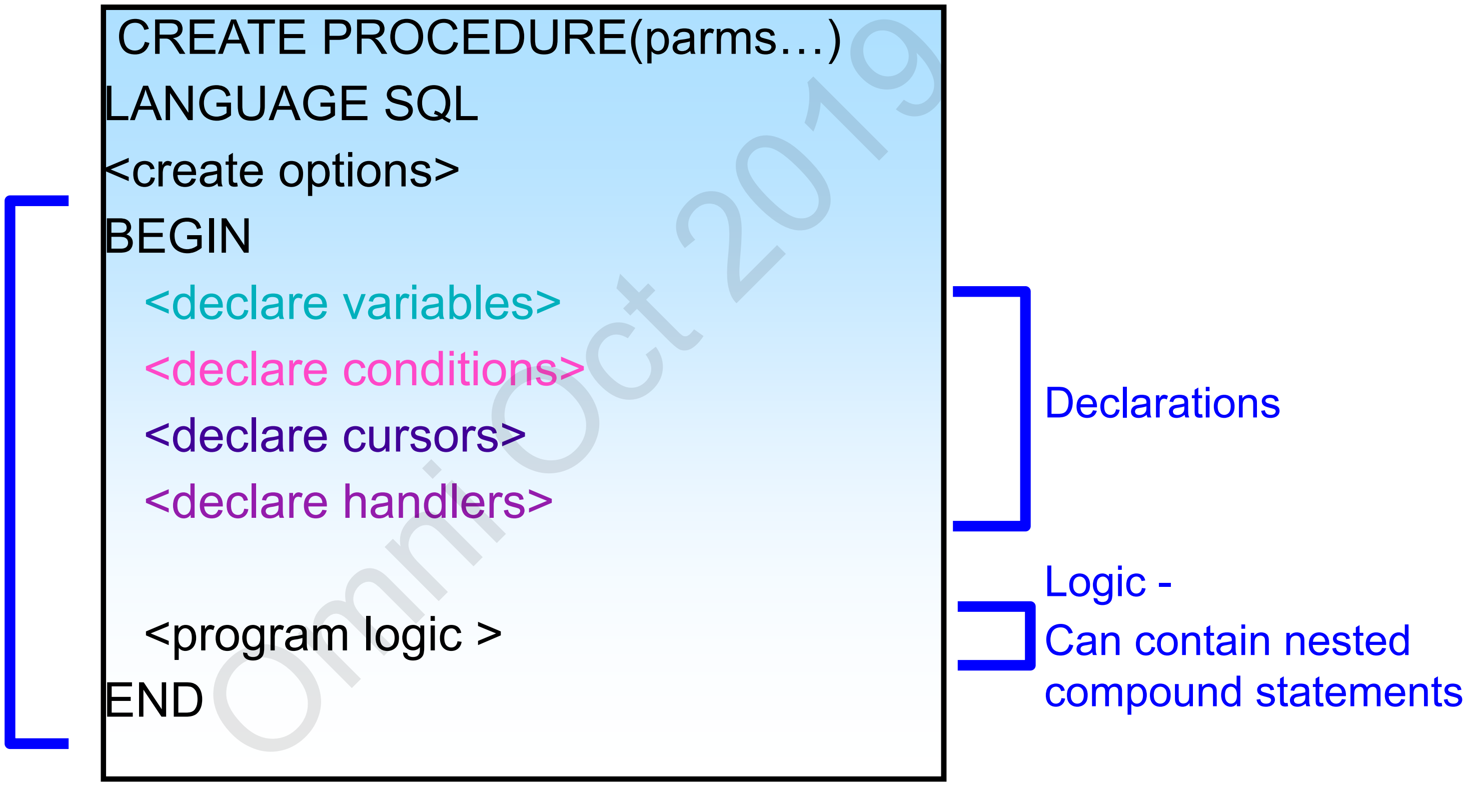
Stored Procedure to encapsulate

- Encapsulate complex logic into a single service
- Reduce traffic flow
  - One request initiates multiple transactions and processes



- Further enhance data flow and reduce work files by returning result sets

General SQL Procedure layout



```
CREATE PROCEDURE(parms…)
LANGUAGE SQL
<create options>
BEGIN
   <declare variables>
   <declare conditions>
   <declare cursors>
   <declare handlers>

   <program logic >
END
```

Basic Procedure (Language) Constructs

- DECLARE – define variables. Variables automatically initialized when procedure is called
- SET – assign a value to a variable or parameter
- SET OPTION – 'compile' options
- Comments - either /* */ or --
- Logic statements
  - IF THEN ELSE END IF
  - CASE
- Looping constructs
  - FOR
  - LOOP
  - REPEAT
  - WHILE
- Error handling and feedback
  - CONDITIONs and HANDLERs
  - GET DIAGNOSTICS
  - SIGNAL and RESIGNAL
  - RETURN
- Result Sets

SQL Procedure example

```
create or replace procedure exitcheck(out totalcount int)
language sql modifies sql data
begin
  declare pgmcount int default 0;
  declare exc_occurred int default 0;
  declare continue handler for sqlexception set exc_occurred = 1;

  set totalcount = 0; /* assume the best */
  Loop1: for Loop1 as c1 cursor for
  with exitlist (entry) as (
   values('QIBM_QDB_OPEN'),('QIBM_QSQ_CLI_CONNECT'))
   select entry from exitlist
  do
   if exc_occurred = 1 then leave Loop1; end if;
   set pgmcount = 0;
   call regcount(Loop1.entry,pgmcount);
   if pgmcount > 0 then
      set totalcount = totalcount + pgmcount;
   end if;
  end for;
  if exc_occurred = 1 then
    set totalcount = -1;
  end if;
end; /* exitcheck */
```

Techniques to Consider

Parameter passing options

Leverage default parameters to lessen the number of procedures needed

Example:

```
create or replace procedure trimfile
   (thelib char(10),
    trimdate date default (current date - 1 year))
language sql
begin
  declare d_sql varchar(3000);
  …
  set d_sql = ‘delete from ‘ concat thelib concat ‘.orders where order_date < ?’;
  prepare stmt1 from d_sql;
  execute stmt1 using trimdate;
end;
```

Invoke as:

call trimfile(‘mylib’,’10/01/2019’)

or

call trimfile(‘mylib’);

Controlling procedure name

Procedures can have long, meaningful SQL names

- There can also be multiple procedures with the same (long) name in the same library, with different number of parameters
- Control corresponding pgm/srvpgm object name to help organize objects
  - Use **SPECIFIC**

Example (assuming the procedures do different things):

```
create or replace procedure my_meaningful_proc_name()
language sql
SPECIFIC MYPROC0 /* control name of object in IBM i library */
…

create or replace procedure my_meaningful_proc_name(parm1 int)
language sql
SPECIFIC MYPROC1 /* control name of object in IBM i library */
…
```

Mixing static and dynamic

Using dynamic SQL within SQL procedures is very powerful

- Build statements based on input and environment situations
- Take advantage of dynamic's 'late binding'

Static – Things you know about during the procedure creation

Dynamic – to handle things that can vary

Use **prepare/execute** and **execute immediate** to drive dynamic

```
create or replace procedure trimfile(thelib char(10), trimdate date) language sql
begin
  declare d_sql varchar(3000);
  …
  Set d_sql = 'delete from ' concat thelib concat '.orders where order_date < ?';
  prepare stmt1 from d_sql;
  execute stmt1 using trimdate;
end;
```

Control compile - SET OPTION

The SET OPTION controls how the procedure is created. Common options:

- Compile for debugging

```
CREATE OR REPLACE PROCEDURE MYPROC(…)
LANGUAGE SQL SET OPTION DBGVIEW = *STMT
```

Note: alternative is to use the **ALLOW DEBUG MODE** clause

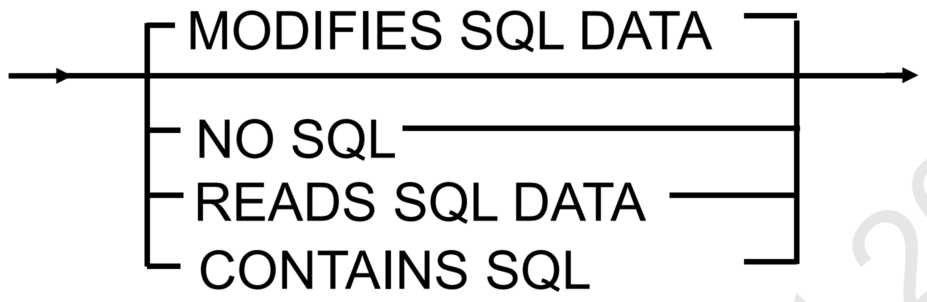
- Commitment control level
  - Specify lowest commit level to accomplish what you need

```
SET OPTION COMMIT = *NC
```

- Target release
  - Specify target release to help ensure program can run at the earliest necessary release.
  - Note this does not ensure dynamic SQL will run at that release!

```
SET OPTION TGTRLS = V7R1M0
```

Create Option – Data access allowed

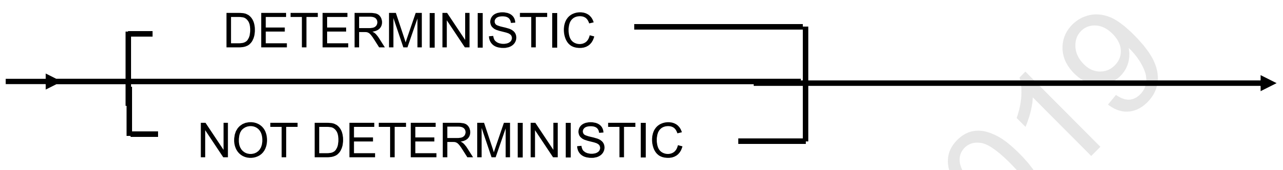


▪ MODIFIES SQL DATA	– Most any SQL statement allowed
▪ READS SQL DATA	– Read Only statements
▪ CONTAINS SQL	– Simple local statements (SET, DECLARE)
▪ NO SQL	– No SQL allowed (external procedures only)

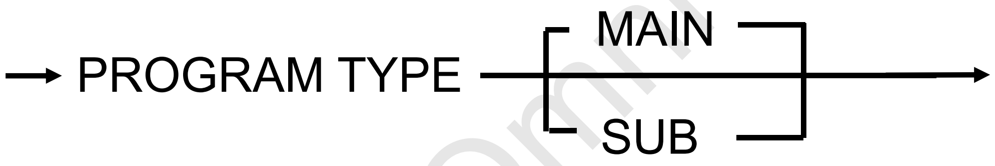
Note: Create routine at the ‘lowest’ option possible for your situation

- Lowest to highest: NO, CONTAINS, READS, MODIFIES

Create Options



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



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

Looping Constructs

For the most part, which looping construct to use is a developer’s choice

- LOOP
- WHILE
- REPEAT

The FOR is a unique solution when processing the rows of a cursor result

- Saves steps of defining a separate cursor and variables
- Allows columns in SELECT statement to be accessed directly!
- Cursor can be used in WHERE CURRENT OF... operation
- Main disadvantage is that the table(s) must be known at create procedure time

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;
```

Leverage DB2 for IBM i services

IBM i services are a great SQL way to get system related information

- Natural integration into an SQL procedure

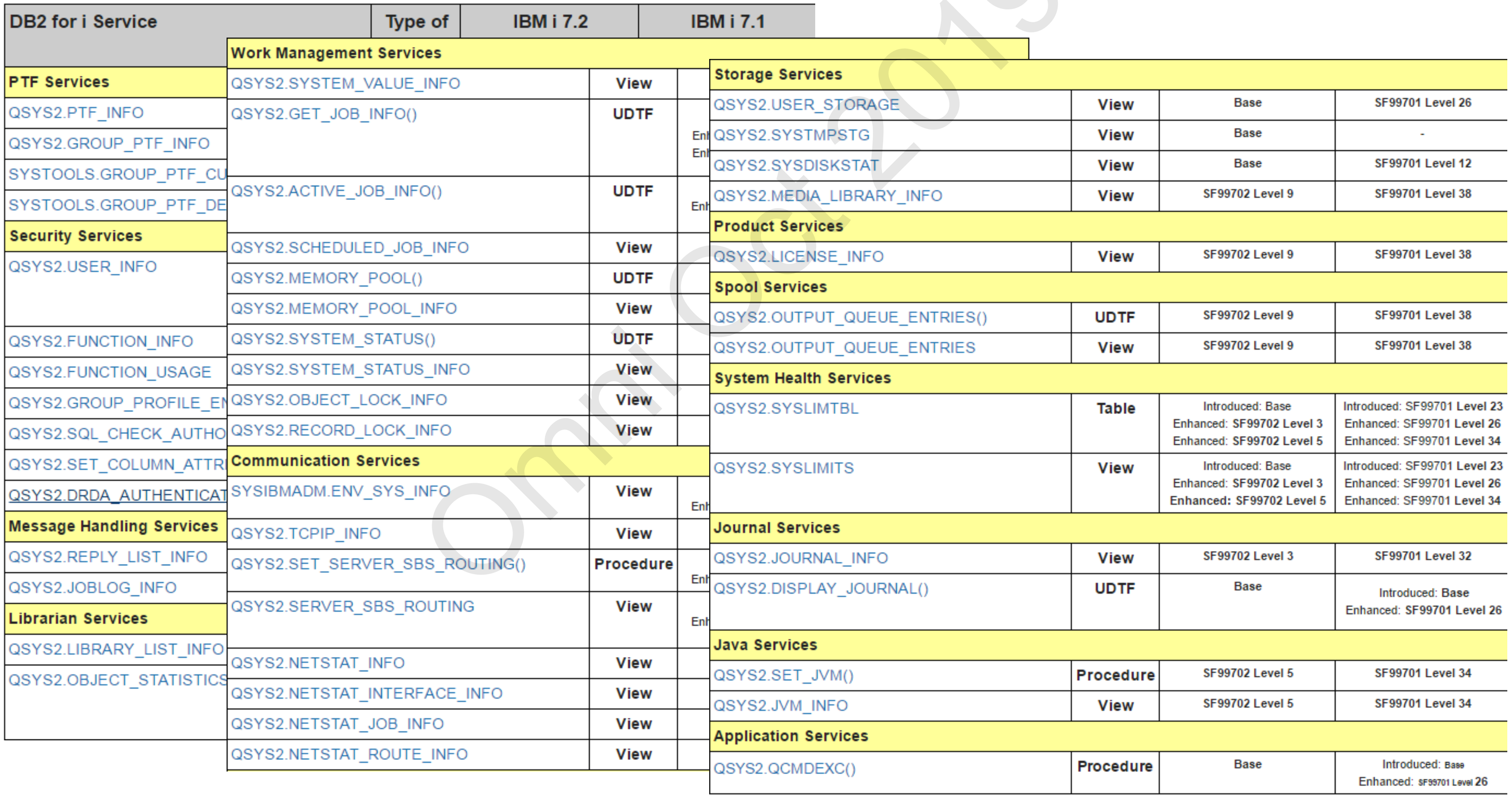
```
create or replace procedure myproc(…)
language sql
begin
  declare exc_occurred int default 0;
  declare host_name varchar(257);
  declare server_ipaddr varchar(45);
  declare vrm varchar(10);
  declare continue handler for sqlexception set exc_occurred = 1;
  …
  set (host_name,vrm, server_ipaddr) =
      (select local_host_name, host_version, server_ip_address
        from qsys2.tcpip_info);

  if vrm = 'V7R1M0' then /* special case to handle i7.1 */
   …
 end if;
end;
```

http://ibm.biz/DB2foriServices

Db2 for IBM i Services

- Complete listing found on IBM i developerWorks: https://ibm.biz/Db2Services
- Service objects found in QSYS2, unless otherwise noted



DB2 for i Service	Type of	IBM i 7.2	IBM i 7.1
PTF Services			
QSYS2.PTF_INFO			
QSYS2.GROUP_PTF_INFO			
SYSTOOLS.GROUP_PTF_CU			
SYSTOOLS.GROUP_PTF_DE			
Security Services			
QSYS2.USER_INFO			
QSYS2.FUNCTION_INFO			
QSYS2.FUNCTION_USAGE			
QSYS2.GROUP_PROFILE_EI			
QSYS2.SQL_CHECK_AUTHO			
QSYS2.SET_COLUMN_ATTR			
QSYS2.DRDA_AUTHENTICAT			
Message Handling Services			
QSYS2.REPLY_LIST_INFO			
QSYS2.JOBLOG_INFO			
Librarian Services			
QSYS2.LIBRARY_LIST_INFO			
QSYS2.OBJECT_STATISTICS			

Work Management Services		
QSYS2.SYSTEM_VALUE_INFO	View	
QSYS2.GET_JOB_INFO()	UDTF	Enl Enl
QSYS2.ACTIVE_JOB_INFO()	UDTF	Enl
QSYS2.SCHEDULED_JOB_INFO	View	
QSYS2.MEMORY_POOL()	UDTF	
QSYS2.MEMORY_POOL_INFO	View	
QSYS2.SYSTEM_STATUS()	UDTF	
QSYS2.SYSTEM_STATUS_INFO	View	
QSYS2.OBJECT_LOCK_INFO	View	
QSYS2.RECORD_LOCK_INFO	View	
Communication Services		
SYSIBMADM.ENV_SYS_INFO	View	Enl
QSYS2.TCPIP_INFO	View	
QSYS2.SET_SERVER_SBS_ROUTING()	Procedure	Enl
QSYS2.SERVER_SBS_ROUTING	View	Enl
QSYS2.NETSTAT_INFO	View	
QSYS2.NETSTAT_INTERFACE_INFO	View	
QSYS2.NETSTAT_JOB_INFO	View	
QSYS2.NETSTAT_ROUTE_INFO	View	

Storage Services			
QSYS2.USER_STORAGE	View	Base	SF99701 Level 26
QSYS2.SYSTMPSTG	View	Base	-
QSYS2.SYSDISKSTAT	View	Base	SF99701 Level 12
QSYS2.MEDIA_LIBRARY_INFO	View	SF99702 Level 9	SF99701 Level 38
Product Services			
QSYS2.LICENSE_INFO	View	SF99702 Level 9	SF99701 Level 38
Spool Services			
QSYS2.OUTPUT_QUEUE_ENTRIES()	UDTF	SF99702 Level 9	SF99701 Level 38
QSYS2.OUTPUT_QUEUE_ENTRIES	View	SF99702 Level 9	SF99701 Level 38
System Health Services			
QSYS2.SYSLIMTBL	Table	Introduced: Base Enhanced: SF99702 Level 3 Enhanced: SF99702 Level 5	Introduced: SF99701 Level 23 Enhanced: SF99701 Level 26 Enhanced: SF99701 Level 34
QSYS2.SYSLIMITS	View	Introduced: Base Enhanced: SF99702 Level 3 Enhanced: SF99702 Level 5	Introduced: SF99701 Level 23 Enhanced: SF99701 Level 26 Enhanced: SF99701 Level 34
Journal Services			
QSYS2.JOURNAL_INFO	View	SF99702 Level 3	SF99701 Level 32
QSYS2.DISPLAY_JOURNAL()	UDTF	Base	Introduced: Base Enhanced: SF99701 Level 26
Java Services			
QSYS2.SET_JVM()	Procedure	SF99702 Level 5	SF99701 Level 34
QSYS2.JVM_INFO	View	SF99702 Level 5	SF99701 Level 34
Application Services			
QSYS2.QCMDEXC()	Procedure	Base	Introduced: Base Enhanced: SF99701 Level 26

Use global variables

Global variables provide several advantages
- communicate across procedures
- define ‘configuration’ or default values
- ‘catch’ OUTput procedure parameters for testing

Example. Use global variable mylib.verbose to control whether to dump out trace information

```
create or replace procedure myproc(…)
language sql
begin
  declare exc_occurred int default 0;
  … handlers here for expected errors…
  …
  if mylib.verbose = 1 then
    … dump out trace type information here to aid debugging
  end if;
end;
```

Calling out

It is sometimes necessary to do a call outside of SQL

- Leverage an OS command
- Get information from outside SQL

QSYS2.QCMDEXC is a good way to do these ‘outside’ calls

```
create or replace procedure myproc(…)
language sql
begin
  …
  CALL QSYS2.QCMDEXC(
     'QSYS/SBMJOB CMD(QSYS/RUNSQL SQL(‘’CALL QGPL.DOSOMETHING()'') ‘
     CONCAT ‘ COMMIT(*NONE) NAMING(*SQL))');        /* submit job to do work */

  CALL QSYS2.QCMDEXC(‘QSYS/DLYJOB DLY(5)’); /* wait for job to get going… */
 …
end;
```

Speaking of CL interaction

- Interaction between native (CL) and SQL is limited when SQL is driving things
  - Errors tend to be generic SQL0443 message
  - often not an issue if simple error handling is sufficient
- If detailed handling is important, you could write your own wrapper to invoke CL
  - Or use an SQL Service against the joblog messages

Example:

```
create or replace procedure myproc() language sql
begin
 declare underlying_error char(7);
 declare exc_occurred int default 0;
 declare continue handler for sqlexception set exc_occurred = 1;
 CALL QSYS2.QCMDEXC(‘QSYS/SBMJOB …’)  /* submit job to do work */
 if exc_occurred = 1 then
   set underlying_error  = (select message_id from
                                table(QSYS2.JOBLOG_INFO('*')) x
                                  where ordinal_position =  (
                                  select max(ordinal_position) - 2
                                  from table(QSYS2.JOBLOG_INFO('*')) x
                                  where message_id = 'SQL0443')
    -- do some acknowledgement of the error….
 end if;
end;
```

Running with elevated authority

In some cases it is advantageous to run a procedure with elevated authority

- when invoked, procedure runs under a higher (pgm owner’s) authority

Authority to call procedure is usually restricted

Note: adopted authority does not work for IFS files

Use SET statement with USRPRF option:

```
CREATE OR REPLACE PROCEDURE myproc(…)
LANGUAGE SQL
SET OPTION DYNUSRPRF = *OWNER, USRPRF = *OWNER
```

Feedback and Error Handling

Procedures can leverage a rich set of error and message handling capabilities

- GET DIAGNOSTICS
- SQLSTATE and SQLCODE variables
- CONDITIONs and HANDLERs
- SIGNAL and RESIGNAL
- RETURN statement

Feedback & Error Handling

- GET DIAGNOSTICS
  - Retrieve information about last statement executed
    - Row_count, return_status, error status….
  - CURRENT or STACKED
    - CURRENT – statement that was just executed
    - STACKED – statement before error handler was entered
      - Only allowed within error handler

Example:

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

Error handling

Every non-trivial procedure, practically speaking, needs error handling

- **DECLARE … HANDLER FOR…**

Get use to having a handler in your ‘template’ procedure to encourage usage

- Use CONDITIONs to convey meaning for cryptic SQLSTATE

```
create or replace procedure myproc(…)
language sql
begin
  declare already_exists int default 0;
  declare alreadyexists condition for '42710';
  declare dupekey condition for '23505';
  declare continue handler for alreadyexists set already_exists=1;
  declare continue handler for dupekey set already_exists=1;
  …
end;
```

General exception handler

It is often useful to have a general exception handler to catch any unexpected errors so as to allow the procedure to finish gracefully

SQL provides a built-in condition called SQLEXCEPTION

```
create or replace procedure myproc(…)
language sql
begin
  declare exc_occurred int default 0;
  … other handlers here for expected errors…
  DECLARE CONTINUE HANDLER FOR SQLEXCEPTION SET exc_occurred = 1;
  …
  IF exc_occurred = 1 then
    -- do some acknowledgement of the error….
  END IF;
end;
```

Returning result sets

Result sets are a unique capability for procedures

- allow an answer set(s) to be returned from the CALL
- Consolidate complex processing for determining an answer set under one CALL
- Communicate much more information back than just a reason code

Example:

```
create or replace stop_processing()
  dynamic result sets 1 language sql modifies sql data
begin
   declare status varchar(50);
   declare total_job_count, job_count int default 0;

   declare cursor1 cursor with return for
     with cte("status", "jobs found", "jobs ended")  as
      (values(status, total_job_count, job_count))
      select * from cte;

    call killjobs(total_job_count, job_count); /* find and kill jobs */
    set status = case when total_job_count = 0 then 'no jobs found'
                      when total_job_count > job_count then 'not all jobs ended'
                      else 'success. All jobs ended' end;
   open cursor1 ; -- cursor left open for client application
end;
```

Constructing a result set

Result sets are a good way to return (conditional) data as a service

Example: return a list of vehicles from optional input filters

```
create or replace procedure get_car_list
  (in g_make varchar(20) default ‘’
   , in g_model varchar(20) default '', in g_year int default 0)
  dynamic result sets 1 language sql modifies sql data
begin
   declare d_sql varchar(1000);
   declare w_clause varchar(500) default ' where';
   declare have_where int default 0;
   declare cursor1 cursor with return for statement1;

   if g_make <> '' then
     set w_clause = w_clause concat ' make = ?';
     set have_where = 1;
   else
     set w_clause = w_clause concat ' "" = ?';
     set g_make = ''; /* make sure not null */
   end if;
   if g_model <> '' then
     set w_clause = w_clause concat ' and model = ?';
     set have_where = 1;
   else
     set w_clause = w_clause concat ' and "" = ?';
     set g_model = ''; /* make sure not null */
   end if;
```

```
 if g_year > 0 then
     set w_clause = w_clause concat ' and year >= ?';
     set have_where = 1;
   else
     set w_clause = w_clause concat ' and 0 = ?';
     set g_year = 0; /* make sure not null */
   end if;

set d_sql =
'SELECT make, model, year, color, style FROM vehicles'
   concat case when have_where <> 0 then w_clause
                     else '' end;

prepare statement1 from d_sql;
/* open cursor can have 'extra' variables */
open cursor1 using g_make, g_model, g_year;
-- cursor left open for client application
end;
```

Result Set Considerations

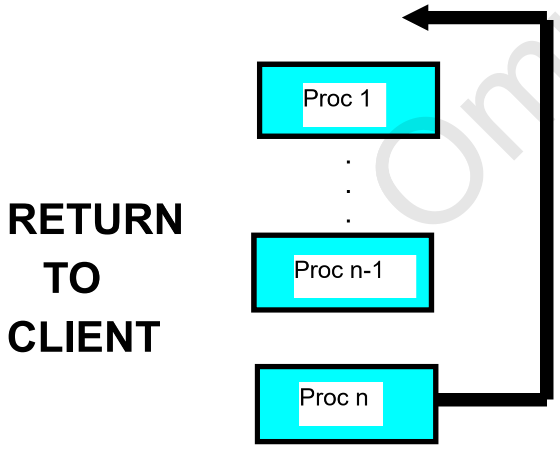
Result Set Consumer Control

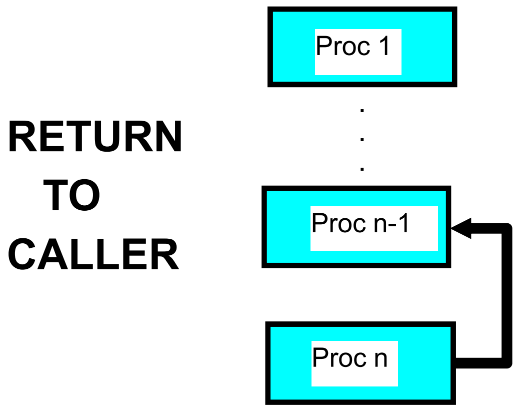
- RETURN TO CLIENT

  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





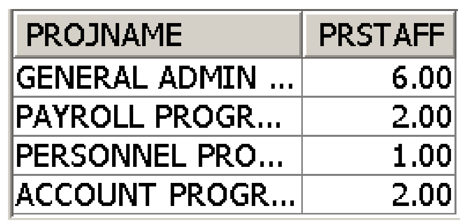
Result Set Consumption

- Consume result sets from one procedure inside another
  - 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;
 ...
```

Consume Result Set!



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

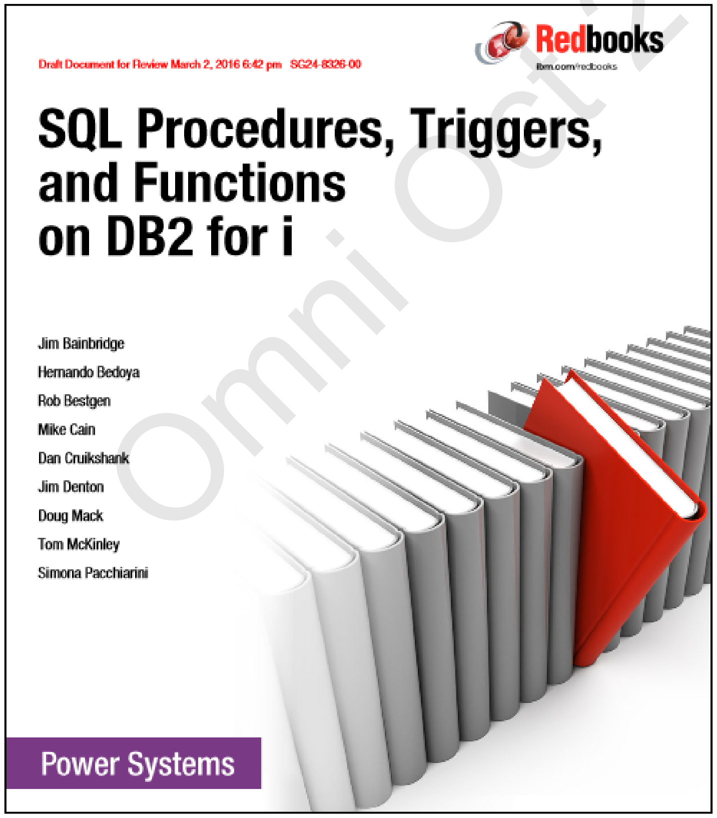
** DESCRIBE PROCEDURE & DESCRIBE CURSOR statements can be used to dynamically determine the number and contents of a result set

More Information

The Full Story

RedBook available

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


Draft Document for Review March 2, 2016 6:42 pm SG24-8326-00

Redbooks
ibm.com/redbooks

SQL Procedures, Triggers, and Functions on DB2 for i

Jim Bainbridge
Hernando Bedoya
Rob Bestgen
Mike Cain
Dan Cruikshank
Jim Denton
Doug Mack
Tom McKinley
Simona Pacchiarini

Power Systems

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

- Customized consulting, education, architecture and design reviews
  - Advanced SQL and Datacentric Programming
  - SQL Performance Best Practices, Monitoring and Tuning
  - RCAC
  - …
- Consulting on any Db2 for i topic!

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

Thank you!

Special notices

Special notices (cont.)

Revised December 2, 2010