Improving SQL procedure performance

New features and tips to help improve performance of SQL Procedures, Functions, and Triggers

Kent Milligan
IBM Technology Expert Labs

January 2025

Abstract

With IBM i 6.1, IBM delivered several performance enhancements that build on the performance improvements first delivered in i5/OS V5R4 with the Expression Evaluator technology, Expression Evaluator was delivered as a new component of Db2 for i designed to enhance the performance of SQL stored procedures, functions, and triggers. This white paper introduces you to Expression Evaluator and the latest performance nhancements. In addition, this paper explains how to determine if your SQL routines are taking advantage of the efficiencies offered by these new capabilities. Also discussed are some nontrivial programming techniques that are aimed at improving the performance your SQL procedural objects.

Introduction

IBM® Db2 for i was the first member of the IBM Db2® family that implemented SQL procedural language (SQL PL). The support for SQL stored procedures was first shipped in IBM OS/400® V4R2. Since then, every new release has delivered a number of enhancements and improved functionality; so over the years, the Db2 for i implementation of SQL PL has matured and become a robust programming-language alternative for SQL and IBM i developers.

The SQL PL, which is based on the ISO/ANSI/IEC SQL Persistent Stored Modules (SQL/PSM) specification, allows developers to write routines (user-defined functions, stored procedures and triggers) that combine SQL access with flow-control structures that are typical for a procedural language. This type of procedural-SQL scripting language has proved to be very popular among database programmers. In fact, all major database vendors offer a version of a scripting language with functionality similar to that of Db2 SQL PL. For example, Oracle supports PL/SQL and Microsoft® and Sybase use Transact-SQL. However, Db2 is the only database that implements a standard compliant-scripting language. Other vendors use proprietary dialects, primarily because they supported SQL procedural languages before the SQL/PSM standard was published.

The Db2 for i support for SQL PL has recently been instrumental in several large porting projects in which hundreds of SQL stored procedures, functions and triggers were successfully ported from other database platforms to Db2 for i. To facilitate these porting efforts, the IBM Rochester development laboratory has shipped a number of significant enhancements that are aimed at improving SQL PL functionality and performance. In this paper, the latest Db2 for i enhancements are highlighted and nontrivial programming techniques are covered — to help increase your SQL PL expertise and improve performance of your SQL procedural objects.

SQL PL implementation methods

On the IBM i platform, when an SQL routine is created, the database internally generates a C program object with embedded SQL to implement the business logic that is described in the procedural SQL object. This C program implementation is transparent to the programmer because Db2 takes care of the program creation and compilation.

If you were to look at this code, you would see a mixture of pure C code and embedded SQL statements (which are converted to system API calls during a precompiler step). In this conversion process, the database uses different code-generation techniques. The technique chosen is dependent on the content of the statement that is being converted. Similar to high-level languages, SQL PL supports statements with standard programming constructs such as loops (FOR, WHILE, REPEAT), conditions (CASE, IF-THEN-ELSE) and assignments (SET varX='TEST').

Pre-V5R4 implementation methods

Prior to i5/OS V5R4, these simple constructs were converted in one of the following methods:

- **Generating pure C code**
  This is the most efficient method because it avoids the overhead of any interaction with the Db2 engine. However, it is only implemented for simple expressions that adhere to certain restrictions (listed in

Appendix A). Usage of this method was expanded with enhancements in IBM i 6.1.

- **Generating a call to the QSQVALUE system module**
  Implemented for those SET statements that contain no expressions. Only SET statements in which literals or special registers are assigned to local variable.
- **Generating an SQL SELECT statement that references the QSQPTABL system table**
  Implemented when neither of the above methods can be used. Statements are converted to underlying queries against the “dummy” system table QSQPTABL in the schema QSYS2. After they are converted, those queries are run to perform the evaluation of the expression.

These methods are listed in the order of their performance characteristic (fastest to slowest). The most expensive of the above implementations is the use of the QSQPTABL system dummy table because of the overhead associated with the open data path (ODP) structure. To illustrate this point, consider the following sample statement that contains an expression taken from an SQL stored procedure:

```
IF (V_SOURCE_VAL IS NULL OR V_TARGET_VAL IS NULL OR V_SOURCE_VAL = V_TARGET_VAL) THEN
```

Prior to i5/OS V5R4, this statement would have been converted to the following embedded SQL query against the QSYS2.QSQPTABL dummy table:

```
SELECT 1 INTO :H FROM QSYS2.QSQPTABL WHERE (H IS NULL OR :H IS NULL OR :H = :H)
```

Although this implementation works well, it does carry the regular overhead associated with SQL statements at runtime. Consider the following steps that occur each time an SQL statement runs on Db2:

1. The SQL statement is parsed.
2. The statement’s access plan is validated and replanned, if necessary.
3. An open data path is created or reused and the cursor is opened.
4. The row is fetched.
5. The cursor is closed.

For SQL procedures with many such expressions to be evaluated, a potentially large number of ODPs can be created over the QSQPTABL table. ODP processing is a processor-intensive activity and requires allocation of temporary storage which increases the memory footprint and slows down the expression evaluation.

Expression Evaluator addition to V5R4 implementation methods

To avoid the performance overhead that is associated with the QSQPTABL method, a fast-path evaluator was needed for expressions and assignments within an SQL procedural object. Expression Evaluator was created to fulfill this need. Introduced in i5/OS V5R4, this enhancement takes those same expression statements and uses a new interface for evaluation and execution. The new expression-evaluator interface trims down both the resources and storage needed to evaluate the expressions and run the statements. When running these expressions and assignments, there is no longer a need for an ODP. The usage of ODPs and cursors are eliminated, thus less processor and memory resources are required. The net result of all this is that the statement (and your procedures) exhibit better performance.

The addition of Expression Evaluator makes the following implementation methods available to Db2 for i when implementing comparison and assignment statements in SQL PL objects.

- Generated C code
- Generated QSQVALUE Call
- Expression Evaluator
- Generating an SQL SELECT statement that references the QSQPTABL system table

Not all expression statements will implement Expression Evaluator. The new code path is not used for scalar SQL statements that have the following characteristics:

- Reference LOB columns or variables
- Reference tables. For example, the following statement would not be implemented with Expression Evaluator: `SET customer_count = (SELECT COUNT(*) FROM customers)`
- Reference user-defined functions (UDFs)

The following table contains examples to better explain when the Expression Evaluator method can be and cannot be used.

Example statement	Statement type	Expression Evaluator used?	Why not?
SET clobvar1 = ‘ABC123’	Assignment	No	CLOB variable reference
SET var1 = ‘ABC’ || ‘123’	Assignment	Yes	-
SET var1 = UPPER(‘abc123’)	Assignment	Yes	-
SET var1 = myUDF1(‘abc123’)	Assignment	No	UDF reference
SET maxval = (SELECT MAX(amt) FROM orders)	Assignment	No	Table Reference
IF v1 = COALESCE(‘ABC’,v2) THEN	Comparison	Yes	-
IF var1 = myUDF2(‘ABC123’) THEN	Comparison	No	UDF Reference
IF UPPER(var1) = ‘ABC123’ THEN	Comparison	Yes	-

Table 1: Expression Evaluator usage table

Expression Evaluator feedback and analysis

The Db2 for i SQL Performance Monitors contain information to help you determine when Expression Evaluator was used in a particular expression statement. In a captured monitor trace, the QVC1E column for the 1000 row type (QQRID) indicates whether Expression Evaluator is used to run the procedural statement. For details on SQL Performance Monitors, see the Redbook *SQL Performance Diagnosis on IBM Db2 Universal Database for iSeries* (SG24-6654) (**ibm.biz**/db2iRedbooks).

The monitor data was enhanced for SQL PL expressions in IBM i 6.1. The possible values for QVC1E are as follows:

- ‘Y’ - Expression Evaluator used
- ‘S’ - Call to QSQVALUE system module
- ‘O’ - Generated SELECT statement opened against QSQPTABL
- ‘N’ - Expression evaluator usage not applicable to the SQL statement (for example, UPDATE t1 SET c1=100)

As mentioned, some simple character assignment statements can be converted to inline C code without the need for dummy table cursors or Expression Evaluator. Although this type of conversion method provides the most efficient code path, those statements are not captured in a database monitor trace since the method does not interact with the Db2 for i engine.

Feedback from the SQL Performance Monitors is currently the only way to determine if Expression Evaluator method is used. You cannot determine the implementation method by analyzing the generated C code — the generated C code can only be analyzed to determine which expressions were implemented with C code.

The following example procedure, *justice_for_all*, was created on two IBM i servers: one at i5/OS V5R3 and the other at i5/OS V5R4. The procedure reads each row in the Employee table and makes adjustments to the value of the employee salary column based on employee tenure and overall average salary. The statements affected by Expression Evaluator are highlighted.

```
CREATE PROCEDURE justice_for_all
  (OUT o_number_of_raises INTEGER, OUT o_cost_of_ranges DECIMAL(9, 2))
   LANGUAGE SQL
  PROGRAM NAME justice

BEGIN
    DECLARE v_avg_tenure, v_number_of_raises INT DEFAULT 0 ;
    DECLARE v_avg_compensation, v_cost_of_raises DECIMAL ( 9 , 2 ) DEFAULT 0 ;

    SELECT AVG ( YEAR ( CURRENT_TIMESTAMP ) - YEAR ( hiredate ) ) ,
         DECIMAL ( AVG ( salary + bonus + comm ) , 9 , 2  INTO v_avg_tenure , v_avg_compensation  FROM employee ;


    FOR EACH_ROW AS c1 CURSOR FOR
        SELECT YEAR ( CURRENT_TIMESTAMP ) - YEAR ( hiredate ) AS tenure ,
                       salary + bonus + comm AS compensation FROM employee
           DO
           IF tenure > v_avg_tenure AND compensation < v_avg_compensation  THEN
               UPDATE employee SET salary = salary + (v_avg_compensation - compensation)
                  WHERE CURRENT OF c1;
               SET v_number_of_raises = v_number_of_raises + 1 ;
               SET v_cost_of_raises = v_cost_of_raises + ( v_avg_compensation - compensation ) ;
           END IF ;
    END FOR ;


    SET o_number_of_raises = v_number_of_raises ;
    SET o_cost_of_raises = v_cost_of_raises ;
END;
```

Figure 1: Procedure justice_for_all

A database monitor trace is started and the procedure is called on each system. Expression Evaluator analysis is performed by running the following SQL statement (see Figure 2) against the monitor data:

```
SELECT qvc1e AS "Exp Eval Used", qqc21 AS "Statement Type", qqc103 AS "Procedure Name", qq1000 AS "Statement"
FROM schema_name.dbmon_table
WHERE qqrid = 1000 AND qqc21 IN ('SV', 'SI', 'VI', 'UP')
ORDER BY qqtime;
```

Figure 2: Expression Evaluator analysis query

Table 2 shows the i5/OS V5R3 analysis results.

Expression Evaluator used	Statement type	Procedure name	Statement
-	SI	JUSTICE	SELECT AVG (YEAR (CURRENT_TIMESTAMP) - YEAR (HIREDATE)) , DECIMAL (AVG (SALARY + BONUS + COMM) , 9 , 2) INTO : H : H , : H : H FROM EMPLOYEE
-	OP	JUSTICE	DECLARE C1 CURSOR FOR SELECT YEAR (CURRENT_TIMESTAMP) - YEAR (HIREDATE) AS TENURE , SALARY + BONUS + COMM AS COMPENSATION FROM EMPLOYEE
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	SI	JUSTICE	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
-	UP	JUSTICE	UPDATE EMPLOYEE SET SALARY = SALARY + (: H : H - : H : H) WHERE CURRENT OF C1
-	SV	JUSTICE	SET : H : H = : H : H + (: H : H - : H : H)

Table 2: i5/OS V5R3 database monitor analysis

Notice that each Expression Evaluator Used column is null. Recall that this information is not captured until i5/OS V5R4. Also, notice all of the SELECT statements against the QSYS2.QSQPTABL dummy table.

Now, compare these results shown in Table 3: i5/OS V5R4 database-monitor analysis.

Expression Evaluator used	**Statement type**	**Procedure name**	**Statement**
N	SI	JUSTICE	SELECT AVG (YEAR (CURRENT_TIMESTAMP) - YEAR (HIREDATE)) , DECIMAL (AVG (SALARY + BONUS + COMM) , 9 , 2) INTO : H : H , : H : H FROM EMPLOYEE
N	OP	JUSTICE	DECLARE C1 CURSOR FOR SELECT YEAR (CURRENT_TIMESTAMP) - YEAR (HIREDATE) AS TENURE , SALARY + BONUS + COMM AS COMPENSATION FROM EMPLOYEE
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
Y	VI	JUSTICE	VALUES (CASE WHEN : H : H > : H : H AND : H : H < : H : H THEN 0 ELSE 1 END) INTO : H
N	UP	JUSTICE	UPDATE EMPLOYEE SET SALARY = SALARY + (: H : H - : H : H) WHERE CURRENT OF C1
Y	SV	JUSTICE	SET : H : H = : H : H + (: H : H - : H : H)

Table 3: i5/OS V5R4 database-monitor analysis

Some stored-procedure assignment statements (including the first two: SET v_number_of_raises = 0 and SET v_cost_of_raises = 0.0) do not appear in either analysis report. The reason for this is because these statements are converted by the database engine into pure C code and recall that such statements are not captured by the SQL Performance Monitors.

What is most interesting is the complete elimination of the SELECT statements against the dummy table in the i5/OS V5R4 trace. Each is replaced by a more efficient “VALUES INTO” statement that implements Expression Evaluator (indicated by the value of “Y” in the “Expression Evaluator Used” column). Because the SELECT statements are eliminated, no ODPs are created for the system dummy table on the i5/OS V5R4 system, which saves processor and memory resources. To demonstrate this savings, an ODP analysis query is run against the collected database monitor data on both systems (see Figure 3):

```
SELECT SUM(qqi6) "Total Time" , COUNT(*) "Nbr Full Opens", qq1000
    FROM COBBG.ZZXX
WHERE  qqrid=1000 AND qqi5=0
    AND qqc21 IN ('OP','SI', 'DL', 'IN', 'UP')
GROUP BY qq1000  ORDER BY 1 DESC;
```

Figure 3: ODP analysis query

The i5/OS V5R3 results of this analysis are shown in Table 4: i5/OS V5R3 ODP analysis.

Total time	**Number of full opens**	**Statement**
56040	1	DECLARE C1 CURSOR FOR SELECT YEAR (CURRENT_TIMESTAMP) - YEAR (HIREDATE) AS TENURE , SALARY + BONUS + COMM AS COMPENSATION FROM EMPLOYEE
29656	1	UPDATE EMPLOYEE SET SALARY = SALARY + (: H : H - : H : H) WHERE CURRENT OF C1
28872	1	SELECT 1 INTO : H FROM QSYS2 . QSQPTABL WHERE : H : H > : H : H AND : H : H < : H : H
8776	1	SELECT AVG (YEAR (CURRENT_TIMESTAMP) - YEAR (HIREDATE)) , DECIMAL (AVG (SALARY + BONUS + COMM) , 9 , 2) INTO : H : H , : H : H FROM EMPLOYEE

Table 4: i5/OS V5R3 ODP analysis

Compare that to the i5/OS V5R4 results shown in Table 5 - i5/OS V5R4 ODP analysis.

Total time	**Number full opens**	**Statement**
51128	1	DECLARE C1 CURSOR FOR SELECT YEAR (CURRENT_TIMESTAMP) - YEAR (HIREDATE) AS TENURE , SALARY + BONUS + COMM AS COMPENSATION FROM EMPLOYEE
29312	1	UPDATE EMPLOYEE SET SALARY = SALARY + (: H : H - : H : H) WHERE CURRENT OF C1
11648	1	SELECT AVG (YEAR (CURRENT_TIMESTAMP) - YEAR (HIREDATE)) , DECIMAL (AVG (SALARY + BONUS + COMM) , 9 , 2) INTO : H : H , : H : H FROM EMPLOYEE

Table 5 - i5/OS V5R4 ODP analysis

Comparison of the two sets of analysis reveals that the QSQPTABL ODP creation for i5/OS V5R3 is not performed for i5/OS V5R4. Again, this is the result of Expression Evaluator's ability to process the request without a table reference to QSQPTABL.

The IBM i Access Client Solution (ACS) SQL Plan Cache tooling is often used for SQL performance analysis. It should be noted that SQL statements using the Expression Evaluator implementation method will not be accessible with that tooling because a query access plan is not required for this method.

Measuring performance improvements

The precise impact of Expression Evaluator on your stored-procedure performance is rather difficult to project, simply because many factors can influence the results. Among the factors are various coding styles – different programming techniques can yield different results. For example, performance varies if the assignment statement is coded in 10 stand-alone statements instead of being coded once in a looping construct that iterates 10 times (more information on this later).

In addition, different performance results occur if the statement is run with the Expression Evaluator but does not take the optimal path through the logic. An example of this is a statement that returns a value requiring subsequent data-type mapping after the expression is evaluated. Consider the following example:

```
DECLARE outVar DECIMAL(5,0);
DECLARE inVar1  INT;
DECLARE inVar2 INT;
SET outVar = inVar1/inVar2;
```

The result of the inVar1/inVar2 expression is an integer. In this case, the Expression Evaluator support must perform additional processing to convert the integer result of the division expression to a decimal value. Simple testing that is similar to what is shown in the above example yields approximately a 15 percent performance improvement when the outVar variable is declared as an integer instead of a decimal.

Consequently, results vary. The best approach is to take your own metrics to determine the potential performance gains in a particular environment. However, in an attempt to get a general idea of what performance improvements might be expected, a series of small benchmarks were conducted. The benchmark results can be referenced in the Appendix A: Expression Evaluator Performance Tests.

Enabling Expression Evaluator

If you have an SQL procedural object that was created on a release prior to IBM i 6.1, then you should recreate those procedural objects to get a C program object generated with the most efficient implementation methods. See the Recreate SQL procedural objects after a new release or Database Group PTF section for additional details.

Tableless Query addition to IBM i 7.1 implementation methods

Starting with the IBM i 7.1 release, the SELECT statement referencing QSQPTABL implementation method was replaced with a more efficient tableless query implementation using the VALUES statement. While it's more efficient, the tableless query implementation still has the overhead of using an ODP.

In summary, these are the methods listed in order of their performance characteristics available to Db2 for i when implementing comparison and assignment statements in SQL PL objects.

- Generated C code
- Generated QSQVALUE Call
- Expression Evaluator
- Generated SQL VALUES statement for tableless query

If a procedural statement still references QSQPTABL in the statement text or Visual Explain plan implementation, then that indicates that the procedural object has not been recreated on a more current Db2 for i release. That procedural object should be recreated to allow Db2 to switch the implementation to the more efficient tableless query implementation.

Tableless Query implementation details

The VALUES statement is used to implement the tableless query method introduced in the IBM i 7.1 release. The use of the VALUES statement can be confusing because you may recall that the VALUES statements is also used with the Expression Evaluator implementation. This overlap reinforces the importance of relying on the QVC1E column in the SQL Performance monitor to determine which of these two implementation methods is used with the VALUES statement.

The Expression Evaluator feedback and analysis section documents the Expression Evaluator implementation for this comparison statement:

```
      IF tenure > v_avg_tenure AND compensation < v_avg_compensation THEN
```

The Expression Evaluator implementation for this comparison was the following VALUES statement.

```
      VALUES (CASE WHEN : H > : H AND : H < : H THEN 0 ELSE 1 END ) INTO : H
```

Even though it's a VALUES statement, the implementation method is Expression Evaluator and the the SQL Performance Monitor data would validate that.

Another way to understand when the VALUES statement is being used in support of Expression Evaluator versus a Tableless Query implementation is to examine the VALUES statement and it examine if it contains any references that are not supported by Expression Evaluator. Recall that the following statement characteristics are not support by Expression Evaluator:

- Reference LOB columns or variables
- Reference tables.
- Reference user-defined functions (UDFs)

Take the following comparison statement as an example:

```
IF var2 > UDF1(v1) THEN
```

This comparison would be implemented with the following VALUES statement:

```
VALUES (CASE WHEN : H > UDF1(: H) THEN 0 ELSE 1 END ) INTO : H
```

The implementation method in this case would be a Tableless Query implementation because the Expression Evaluator implementation does not support UDF references.

Tableless Query analysis

With the 7.1 implementation change, the ‘O’ value meaning for the QVC1E column was updated to reflect that the implementation utilizes an ODP associated with a tableless query or a query referencing QSQPTABL.

- ‘Y’ - Expression Evaluator used
- ‘S’ - Call to QSQVALUE system module
- ‘O’ – Expression handled by a query open data path(ODP)
- ‘N’ - Expression evaluator usage not applicable to the SQL statement

Since the Tableless Query implementation does require a query access plan, its associated VALUES statements can be analyzed using the ACS SQL Plan Cache and Visual Explain tooling.

Performance tips for procedures, triggers and functions

This section provides some optimization tips and techniques to help improve the performance of your SQL procedures, triggers and functions. These tips are independent of the Expression Evaluator enhancement and can improve performance on earlier releases except where noted. For a more comprehensive list of performance tips, refer to the IBM manual *Db2 for i SQL Programming*, section “Improve performance of procedures and functions.” All the Db2 for i manuals can be accessed online (**ibm.**biz/db2iBooks).

One of the key recommendations for optimal performance is that programmers use the same data type and lengths on any comparison or assignment statements within an SQL procedure, trigger, or function.

SQL PL performance tips examined in this section:

- Avoid single-statement stored procedures
- Utilize service program objects (beginning with IBM i 6.1 release)
- Minimize the number of calls to other SQL stored procedures
- Move handlers for specific conditions and statements within a nested compound statement
- Combine sequences of complex SET statements into one statement
- Avoid using temporary variables
- Use integer data type instead of character for simple flags
- Use integer data types instead of decimal with zero scale
- Use character data type over variable-length type
- Deconstruct complex IF statements
- Use IF statement instead of COALESCE function
- Specify CCSID 65535 for character variables when using IBM i Access Client Solutions Runs SQL Scripts
- Recreate SQL procedural objects after new IBM i releases and Database Group PTFs

Avoid single-statement stored procedures

One of the primary benefits of stored procedures is their ability to help a client-server applications reduce trips across the network.

Consider the following steps that occur each time the client makes a database request to the server,

- The request flows from the client, over the network, to the server.
- The database engine processes the database request.
- The results set and return codes flow from the server over the network, back to the client.

For example, a Java™ client application issues five JDBC calls that are interlaced with some business logic, resulting in 10 network trips back and forth from the client and server, As an alternative, the database requests and business logic can be moved to a stored procedure and the Java application can make only one call to that procedure. Because processing remains on the server for the duration of the stored procedure, this implementation reduces the number network trips from 10 to 2.

For this reason, procedures are most effective from a performance perspective when multiple operations are performed on a single procedure call. Coding single-statement stored procedures eliminates this advantage because the network trips are not reduced. In fact, such an implementation is clearly not advisable, because SQL stored-procedure calls are unbound calls to program objects on the IBM i platform and the additional program-stack overhead required degrades performance.

Utilize service program objects

Db2 for i generates a C program object by default when implementing SQL routines. In IBM i 6.1, the ability to create a C service program object was added for SQL stored procedures. This is done by specifying the PROGRAM TYPE SUB clause in the SQL procedure source as highlighted in Figure 11. The default program type is MAIN which results in a regular program object being generated.

```
CREATE PROCEDURE ADD_SRVPGM (IN p1 INT, IN n INT, OUT o1 INT)
LANGUAGE SQL
PROGRAM TYPE SUB
BEGIN
  DECLARE v1 INT;
  SET v1=ABSVAL(n);
  SET o1= p1+v1;
END;
```

Figure 4: PROGRAM TYPE SUB example

The usage of a service program object will provide a small performance boost when the SQL stored procedure is called. No changes are needed to the applications that invoke the SQL stored procedure. The PROGRAM TYPE SUB clause is not supported for SQL Triggers or Functions.

Minimize calls to other SQL stored procedures

As mentioned, Db2 for i implements SQL routines as C programs with embedded SQL statements. When the stored procedure is called from an SQL interface, the C program is started as an unbound call in the program stack. If the stored procedure calls yet another stored procedure (a nested call), another unbound program is started in the program stack. The unbound nature of these calls creates additional overhead and degradation in performance. Therefore, when optimal stored-procedure performance is critical, and you believe that your stored procedures do not achieve the required performance metrics, minimizing nested procedures calls is one optimization technique to consider.

Move handlers for specific conditions and statements into nested compound statements

In this section, the benefits of defining condition handlers in nested compound statements are examined. A condition handler is a statement in an SQL stored procedure that is run when an exception or completion condition occurs within the body of a compound statement. The action specified in a handler can be any SQL statement, including another compound statement. The scope of a handler is limited to the compound statement in which it is defined.

For each database operation in an SQL stored procedure, code is generated to determine if each of the handlers that are declared within that compound statement need to be called. When the handlers are declared at the mainline level, it becomes a global handler, and every database-operation statement in the entire stored procedure must generate code for each of the global handlers that are defined. For example, using this programming style, a global handler that is intended to handle a **not found** condition for a DELETE statement also generates code to check for this condition when an INSERT statement is run. This additional processing uses system resources and impacts performance when it is not essential.

This unnecessary processing can be minimized by moving handlers and their associated SQL statements within a nested compound statement. Because the handler is scoped to the compound statement in which it is defined, code is generated only for handlers within that compound statement.

Consider the example shown in Figure 12 where the handler is defined at the mainline level:

```
BEGIN
   DECLARE CONTINUE HANDLER
                 FOR SQLSTATE ' 23504'...
   ...
   DELETE FROM master WHERE id=1;
   ...

BEGIN
   ...
   BEGIN
        DECLARE CONTINUE HANDLER FOR
                                      SQLSTATE ' 23504'...
        DELETE FROM master WHERE id=1;
    END
   ...
```

Figure 12: Example of code that contains a handler that is defined at the mainline level

Combine sequences of complex SET statements into one statement

This particular tip improves performance only if individual SET statements are **not** eligible for C-code generation. The reason for the performance improvement is that assignments can be packaged together and a single invocation to the implemented conversion interface can be made. When the statements are separate, this packaging is not done and the conversion method must start the interface once for each statement. In the cases where C code is generated, each assignment is converted to an individual C statement; therefore, there is no packaging and no benefit gained.

For example, Figure 13 contains sample code with separate SET statements:

```
SET var1 = inp1 * 15;
SET var2 = SUBSTR(inp2,1,5);
SET var3 = RTRIM(inp3) || RTRIM(inp4);
```

Figure 5: Example of separate SET statements.

These three SET statement can be rewritten into one statement (see Figure 14):

```
 SET var1 = inp1 * 15, var2 = SUBSTR(inp2,1,5), var3 = RTRIM(inp3) || RTRIM(inp4);
```

Figure 6: SET statements combined into one statement.

Testing of the single SET statement yields the results shown in Table 6: Results of combining sequences of complex SET statements into one statement:

Execution (within same job)	Average runtime improvement after change was made
1	48.15 percent
2	50.44 percent
3	50.13 percent
4	49.30 percent

Table 6: Results of combining sequences of complex SET statements into one statement

Avoid using temporary variables

Similar to the previous tip, you can reduce invocations to the implemented conversion method by eliminating temporary work-variable assignments and combining the ultimate result statement into one long expression. Both the declaration and assignment overhead of the temporary variables can be eliminated.

For example, Figure 15 shows a stored procedure that uses temporary variables:

```
DECLARE counter INTEGER;
DECLARE var1 INTEGER;
DECLARE var2 INTEGER;
DECLARE var3 INTEGER;
DECLARE var4 INTEGER;

SET counter = 0;
xLoop:
REPEAT
  SET counter = counter + 1;
  SET var1 = counter;
  SET var2= counter * 7;
  SET var3 = counter * 30;
  SET var4 = var1 + var2 + var3;
UNTIL counter = 1000

END REPEAT xLoop;
```

Figure 15: Stored-procedure example using temporary variables.

The large number of DECLARE and SET statements in Figure 15 can be rewritten so that the temporary variables are removed (see Figure 16):

```
DECLARE counter INTEGER;
DECLARE var4 INTEGER;

SET counter = 0;

xLoop:
REPEAT
  SET counter = counter + 1;
  SET var4 = counter + (counter * 7) + (counter*30);
  UNTIL counter = 1000
END REPEAT xLoop;
```

Figure 7: Temporary variables are removed.

Testing of the recommended enhancement yields the results shown in Table 7:

Execution (within same job)	Average runtime improvement after change was made
1	63.68 percent
2	62.85 percent
3	63.04 percent
4	62.99 percent

Table 7: Results of removing temporary variables

Use Integer data type instead of Character for simple flags

Variables declared with Character data types require additional overhead because the database engine must perform more processing and validation. Calls to the engine can be avoided if the Integer data type is used and the statement can be converted to simple C code. Obviously, there are times when usage of the Character type is necessary cannot be avoided. However, it is advisable to look for logic where Integer data types can be used, instead. Good candidates for this type of conversion are variables defined as characters that are used as flags or indicators (values of **0** or **1**).

For example, the code in Figure 17 uses Character data types instead of Integer data types.

```
DECLARE  counter INTEGER ;
DECLARE var1 CHAR(1);
DECLARE var2 CHAR(1);
DECLARE var3 CHAR(1);

SET counter = 0;
xLoop:
REPEAT
  SET counter = counter + 1;
  SET var1 = '1';
  SET var2 = '0';
  SET var3 = '2';
UNTIL counter = 1000
END REPEAT xLoop;
```

Figure 17: Stored-procedure example using character data types instead of Integer data types

However, this stored procedure can be rewritten such that the Character data types are changed to Integer data types (see Figure 8: Character data types changed to Integer data types).

```
DECLARE  counter INTEGER ;
DECLARE var1 INTEGER;
DECLARE var2 INTEGER;
DECLARE var3 INTEGER;

SET counter = 0;
xLoop:
REPEAT
  SET counter = counter + 1;
  SET var1 = 1;
  SET var2 = 0;
  SET var3 = 2;
UNTIL counter = 1000
END REPEAT xLoop;
```

Figure 8: Character data types changed to Integer data types

Testing of the example in Figure 8 yields the results shown in Table 8:

Execution (within same job)	Average runtime improvement after change was made
1	42.37 percent
2	42.05 percent
3	42.15 percent
4	42.08 percent

Table 8: Results of avoiding the use of Character or Date data types

Use Integer data types instead of Decimal with zero scale

Decimal and Numeric data types require more overhead because the database engine must test and handle overflow conditions. Again, Integer data types have a distinct performance advantage when C code can be generated. The improved performance is especially evident for a variable used as a counter. When the variable is incremented by one, efficient C code can be generated to perform the operation.

For example, the code in Figure 19 uses Decimal data types instead of Integer data types.

```
declare counter1 decimal (5,0);
declare counter2 decimal (5,0);
declare counter3 decimal (5,0);

SET counter1 = 0;
xLoop:
REPEAT
  SET counter1 = counter1 + 1;
  SET counter2 = counter2 + 1;
  SET counter3 = counter3 + 1;
until counter1 = 10000
END REPEAT xLoop;
```

Figure 19: Stored-procedure example using Decimal data types instead of Integer data types

However, this stored procedure can be rewritten Integer data types (see Figure 20).

```
declare counter1 integer;
declare counter2 integer;
declare counter3 integer;

SET counter1 = 0;
xLoop:
REPEAT
  SET counter1 = counter1 + 1;
  SET counter2 = counter2 + 1;
  SET counter3 = counter3 + 1;
until counter1 = 10000
END REPEAT xLoop;
```

Figure 20: Decimal data types changed to Integer data types

When the sample is rewritten to use Integer data-type, performance improves dramatically. In fact, rather than percentages, the test results shown in Table 9 are reflected in times X improvement.

Execution (within same job)	**Average runtime improvement after change was made (times X)**
1	109.75
2	153.43
3	153.22
4	153.50

Table 9: Results of using Integer data types instead of Decimal with zero scale

Usage of integer type for counting variables also enables Db2 to use generated C code when an integer variable is incremented by 1 or decremented by 1.

The Decimal data type should also be used (instead of Numeric) for variables that require greater-than-zero scale. The Numeric data type is the SQL implementation of Zoned Decimal and is not natively supported by the IBM i C compiler. This is relevant because Db2 for i implements SQL PL objects as C-program objects. When it encounters Zoned Decimal fields or variables, the C compiler performs internal data-type mapping routines to convert them to the supported packed-decimal data type (which is implemented as Decimal in SQL PL). As with all non-optimal code paths, this mapping comes at the expense of performance; the only way to avoid it is to avoid specifying Numeric data types.

Utilize the Character data type over Variable-Length Character type

Db2 for i can generate and use C code more often for fixed-length character variables than it can for variables declared with the variable-length character data type (VARCHAR). Additional usage of C code occurs most often the variables are referenced in the comparison clause of an IF statement.

For example, the code in Figure 21 uses the VARCHAR data types instead of the fixed-length character type (CHAR). Db2 is able to use generated C code to implement the first IF statement because the VARCHAR variable is being compared with a literal string. However, the second IF statement cannot be implemented with C code when the VARCHAR variable is compared with another variable.

```
DECLARE v1 VARCHAR(10) DEFAULT 'var1';
DECLARE v2 VARCHAR(10) DEFAULT 'var2';
DECLARE v3 INTEGER;

IF v1 = 'abcd' THEN
  SET v3=2;
END IF;


IF v1 = v2 THEN
  SET v3=2;
END IF;
```

Figure 21: Stored-procedure example using character data types instead of Integer data types

Performance of this code can be improved by switching both variable definitions to the CHAR data type allowing Db2 to use generated C code for both IF statements. Figure 22 contains an improved version of the code. Notice that the literal string on the first IF statement had to be padded with blanks to make the literal string the same length as the variable (v1) being compared on the IF check. When a fixed-length character variable is being compared with a literal string, generated C code can only be used when the variable and literal string are the same length.

```
DECLARE v1 CHAR(10) DEFAULT 'var1';
DECLARE v2 CHAR(10) DEFAULT 'var2';
DECLARE v3 INTEGER;

IF v1 = 'abcd      ' THEN
  SET v3=2;
END IF;


IF v1 = v2 THEN
  SET v3=2;
END IF;
```

Figure 9. Stored-procedure example using character data types instead of Integer data types

Deconstruct complex IF statements

Converting a complex IF statement into multiple, nested IF statements can improve the performance of SQL routines. Complex IF statements that contain multiple test conditions (IF (x=10) AND (y=500)) are not eligible to be implemented with generated C code. Breaking a complex IF statement into nested, single-condition IF statements provides more opportunities for Db2 to use generated C code in the runtime implementation. Obviously, this deconstruction approach might not be possible or feasible in all situations, but it should be considered when trying to meet performance requirements.

Notice that sample code in Figure 9 contains two complex IF statements. The first IF statement ORs two conditions together while the second IF statement uses the AND operator on it's two test conditions. Multiple test conditions means that Db2 is unable to use generated C code to implement the IF checks — instead a more expensive implementation must be used by Db2.

```
DECLARE v1 BIGINT;
DECLARE v2 INTEGER;
DECLARE val_size SMALLINT;
…
IF ((v1=1) OR (v2 = 1)) THEN
  SET val_size=0;
END IF;
IF ((v1=10) AND (v2=10)) THEN
   SET val_size=1;
ELSE
   SET val_size = 2;
END IF;
…
```

Figure 10. Stored procedure example using complex IF statements

Improving the runtime performance of the IF checks can be accomplished by deconstruction the multiple-condition IF statements into single-condition IF statements as demonstrated in Figure 11. Db2 implements all of the IF statements in this second example using generated C code. Notice that the deconstruction of the complex IF statements results in the expression on the THEN or ELSE legs being duplicated in the source. Thus, you should take the size of the THEN/ELSE leg expression into consideration when determining whether or not to use this technique.

```
DECLARE v1 BIGINT;
DECLARE v2 INTEGER;
DECLARE val_size SMALLINT;
…
/* Deconstruction of OR condition */
IF (v1=1) THEN
  SET val_size=0;
END IF;
IF (v2=1) THEN
  SET val_size=0;
END IF;


…
```

Figure 11. Stored procedure example using single-condition IF statements

Replace COALESCE functions invocations with IF statements

SQL routines that accept null-capable parameters often use the Coalesce function to replace a null parameter value with a real value. This technique is demonstrated in Figure 12. Although the Coalesce function provides a simple method for replacing null values, the runtime implementation is slower than generated C code.

```
CREATE PROCEDURE proc1
  (IN p1 INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE v1 INTEGER;

  SET v1 = COALESCE(p1, 100);
...
END;
```

Figure 12. Stored procedure example using Coalesce function

To enable the usage of generated C code, the Coalesce function can be converted into an IF statement as shown in Figure 13. Db2 can implement the IF statement with generated C code resulting in improved runtime performance.

```
CREATE PROCEDURE proc1
  (IN p1 INTEGER)
LANGUAGE SQL
BEGIN
  DECLARE v1 INTEGER;

  IF p1 IS NULL THEN
    SET p1 = 100;
  ELSE
     SET v1 = p1;
  END IF;
...
END;
```

Figure 13. Stored procedure example using IF statement to simulate Coalesce function

Specify CCSID 65535 for character variables when using IBM i Access Client Solutions Runs SQL Scripts

Simple character assignments can be implemented through the C-code generation method if the variable CCSID is the same as the source CCSID, or either CCSID is 65535. If neither of these conditions exists, the QSQVALUE module is called to perform these types of assignments and typically does not carry out the task as efficiently as C code does. As mentioned, stored procedures that are designed to enable the C-code generation method run faster than those that do not.

If the IBM i Access Client Solutions (or System i Navigator) Run SQL Scripts tool is your SQL stored-procedure development tool of choice, you must declare your character host variables as CCSID 65535 to obtain this behavior. A CCSID of 65535 indicates binary data that is not to be converted. Run SQL Scripts is a Unicode-based interface and, as such, has conditions that currently prevent matching the variable CCSID with the source CCSID. As a result, when using this interface, specifying CCSID 65535 is the only current way to establish the proper environment for C-code generation of simple character assignments and comparisons.

Proceed with caution

Assigning a CCSID of 65535 to a variable means that the value is never translated on assignments and comparisons. Therefore, you should specify CCSID 65535 if the variable is internal to the procedure and is only used for assignments and comparisons that are part of decision-making processes in the procedure. Avoid specifying this CCSID for variables that might eventually be used in procedure statements that update or insert rows in the database.

For example, the code shown in Figure 14 declares Character data types without CCSID 65535.

```
DECLARE counter INTEGER
DECLARE var1 CHAR(10);
DECLARE var2 CHAR(10);
DECLARE var3 CHAR(10);

SET counter = 0;
xLoop:
REPEAT
  SET counter = counter + 1;
  SET var1 = 'ABCDEFG';
  SET var2 = 'HIJKLMNOP';
  SET var3 = 'QRSTUV';
until counter = 1000
END REPEAT xLoop;
```

Figure 14. Stored-procedure example that declares Character data types without CCSID 65535

However, this Character data types can be defined with CCSID 65535 (see Figure 15).

```
DECLARE counter INTEGER
DECLARE var1 CHAR(10) CCSID 65535;
DECLARE var2 CHAR(10) CCSID 65535;
DECLARE var3 CHAR(10) CCSID 65535;
SET counter = 0;
xLoop:
REPEAT
  SET counter = counter + 1;
  SET var1 = 'ABCDEFG';
  SET var2 = 'HIJKLMNOP';
  SET var3 = 'QRSTUV';
until counter = 1000
END REPEAT xLoop;
```

Figure 15. Character data types that are defined with CCSID 65535

Notice that the only changes to the stored procedure are the additions of the CCSID to the Character variable declarations. In Figure 21, the SET statements for the Character host variables are converted through the QSQVALUE module. When this sample code is rewritten as shown in Figure 22, those statement are converted to pure code and a significant improvement increase can be observed. Again, the test results shown in Table 10: are reflected in times X improvement rather than percentages.

Execution (within same job)	Average runtime improvement after change was made (times X)
1	11.37
2	12.86
3	12.88
4	12.57

Table 10: Results of specifying character variables as CCSID 65535

Recreate SQL procedural objects after a new release or Database Group PTF

New release of the IBM i operating system and new Database Group PTFs (**ibm.com**/ibmi/techupdates/Db2) often contain improvements to the internal implementation of the SQL procedures, functions, and triggers. Frequently these performance improvements can only be realized by re-generating the C program object associated with your SQL PL object.

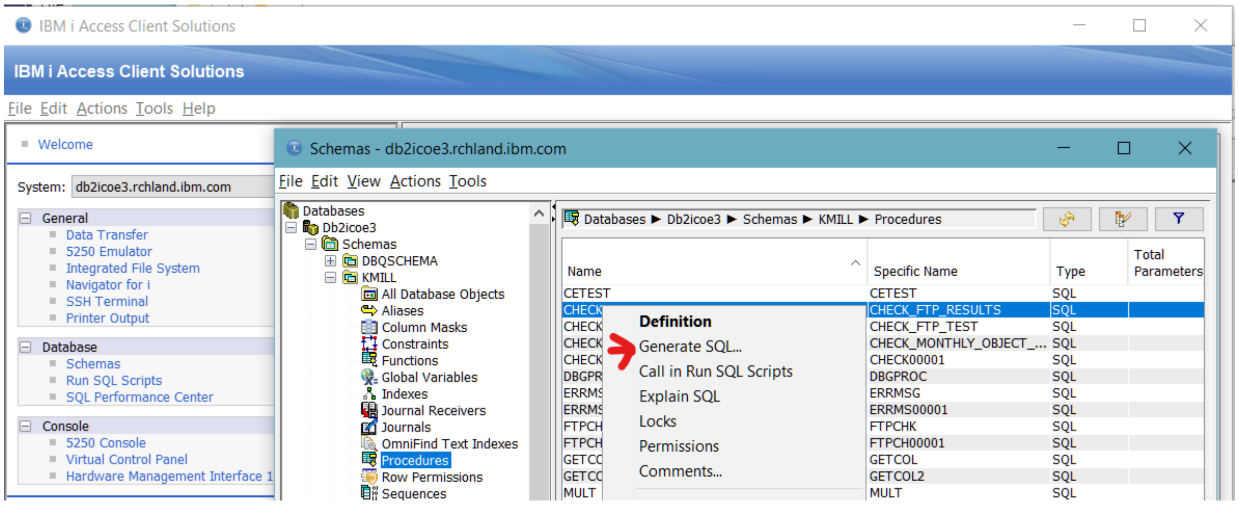
The simplest way to re-generate the C program object for SQL procedures and functions is to use the ALTER statement. As you can see from the following examples, the ALTER statement enables the program object to be recreated without having to include all of the SQL source code.

- ALTER PROCEDURE my_procedure ALTER LANGUAGE SQL;
- ALTER PROCEDURE my_function ALTER LANGUAGE SQL;

Another benefit of using the ALTER statement is that it preserves all existing authorities and privileges of the specified procedure and function.

The only way to regenerate the C program object for an SQL Trigger is to re-run the original CREATE TRIGGER statement.

If you need assistance retrieving the original CREATE statement, the Generate SQL feature in ACS greatly simplifies the task of recreating SQL procedural objects. From the ACS Schemas function, navigate to the Procedures folder of the desired Schema and simply select all the stored procedures you want to recreate, then right-click and select Generate SQL from the menu (as shown in Figure 16: Generate SQL for stored procedures).



Name	Specific Name	Type	Total Parameters
CETEST	CETEST	SQL	
CHECK	CHECK_FTP_RESULTS	SQL	
CHECK	CHECK_FTP_TEST	SQL	
CHECK	CHECK_MONTHLY_OBJECT_...	SQL	
CHECK	CHECK00001	SQL	
DBGPR	DBGPROC	SQL	
ERRMS	ERRMSG	SQL	
ERRMS	ERRMS00001	SQL	
FTPCH	FTPCHK	SQL	
FTPCH	FTPCH00001	SQL	
GETCO	GETCOL	SQL	
GETCO	GETCOL2	SQL	
MULT	MULT	SQL	

Figure 16: Generate SQL for stored procedures

Summary

The objective of this white paper is to give you a better understanding of the more advanced capabilities of Db2 SQL PL, as well provide some performance considerations. This robust and mature implementation of the SQL-based programming language can significantly reduce the cost of database porting projects as well as facilitate modernization of existing IBM i applications. IBM intends to continue to provide exciting functional and performance SQL PL improvements in future releases of IBM i.

Appendix A: Expression Evaluator Performance Tests

For these performance benchmarks, an IBM i system with two identical, dedicated and capped logical partitions is used. One partition was at i5/OS V5R3, the other at i5/OS V5R4 and the tests are run on each. The stored procedures written for the test environment are simple; however, the goal is to test procedures with multiple stand-alone statements and those that use looping constructs. The procedure examples are written solely to test the performance implications of Expression Evaluator, they contain no table references that might otherwise skew the results. For each scenario, seven tests were conducted from the ACS Run SQL Scripts interface. Each test calls the procedure four times. The average runtime-performance improvements are documented in the next several pages.

Performance Benchmark Tests

Scenario 1: Loop with assignments

For the first scenario, the procedure contains a loop construct with 500 iterations. Within each loop iteration, 17 SET statements are run. The purpose of this scenario is to measure the impact of Expression Evaluator within a tight-looping construct that contains assignment statements.

Stored-procedure source code (scenario 1)

Statements shown in blue are implemented with Expression Evaluator in i5/OS V5R4 (see Figure 17).

```
CREATE PROCEDURE Loop500_Assignments()
LANGUAGE SQL
 BEGIN

DECLARE var1 char(3);
DECLARE var2 char(1);
DECLARE var3 char(5) ccsid 65535;
DECLARE var4 char(10) ccsid 65535;
DECLARE counter INTEGER;
DECLARE randomNumVar DECIMAL;

SET counter = 0;
xLoop:
REPEAT

  SET counter = counter + 1;
  SET var1 = 'A  ';
  SET var2 = TRIM(var1);
  SET var3= 'EVEN: ';
  SET var4 = var3 || CHAR(counter);
  SET randomNumVar = RAND() * 100;
  SET randomNumVar = 100;
  SET var3= '>50: ';
  SET var4 = var3 || CHAR(randomNumVar);
  SET randomNumVar = RAND() * 100;
  SET var4 = var3 || CHAR(randomNumVar);
  SET randomNumVar = RAND() * 100;
  SET var4 = var3 || CHAR(randomNumVar);
  SET randomNumVar = RAND() * 100;
  SET var4 = var3 || CHAR(randomNumVar);
  SET randomNumVar = 100;
  SET var4 = var3 || CHAR(randomNumVar);
UNTIL counter = 500
END REPEAT xLoop;
END;
```

Figure 17: Stored-procedure source code (scenario 1)

Test results and analysis (scenario 1)

Database-monitor analysis reveals that, of the 17 assignment condition statements (for each loop iteration), 11 are implemented with Expression Evaluator. Table 11 shows the effect of ODP reuse within a loop. For the i5/OS V5R3 QSQPTABL method, ODP reuse occurs for the statement after the second iteration of the loop on the stored procedure's **first** invocation. The remaining loop iterations (3 through 500) of invocation 1 and all iterations of invocations 2, 3 and 4 benefit from this. Thus, when ODP-reuse mode is in place, the performance improvement in invocations 2, 3 and 4 of the stored procedure is not as profound — but the performance impact for invocation 1 is still quite evident, because Expression Evaluator eliminates the ODP overhead. In many cases, a stored procedure is only called once within a database connection; therefore, the performance impact of Expression Evaluator on the first invocation of an SQL procedure can be significant.

Execution (within same job or connection)	**Average i5/OS V5R4 runtime improvement**
1	57.11 percent
2	18.84 percent
3	14.82 percent
4	16.60 percent

Table 11: Results for scenario-1 testing

Scenario 2.a: Loop with conditions and assignments

For this scenario, the procedure contains a loop construct with 500 iterations. Within each loop iteration, one set of IF-THEN-ELSE statements and five SET statements are run. The purpose of this scenario is to measure the impact of Expression Evaluator within a tight-looping construct that contains both conditions and assignment statements.

Stored procedure source code (scenario 2.a)

Statements shown in blue are implemented with Expression Evaluator in i5/OS V5R4 (see Figure 18).

```
CREATE PROCEDURE Loop500_Conditions_and_Assignments()
LANGUAGE SQL

BEGIN
DECLARE var1 CHAR(3);
DECLARE var2 CHAR(1);
DECLARE var3 CHAR(5) ccsid 65535;
DECLARE var4 CHAR(10) ccsid 65535;
DECLARE counter INTEGER;
SET counter = 0;
xLoop:
REPEAT
  SET counter = counter + 1;
  SET var1 = 'A  ';
  SET var2 = TRIM(var1);
  IF MOD(Counter,2) = 0
  THEN
    SET var3= 'EVEN: ';
    SET var4 = var3 || CHAR(counter);
  ELSE
    SET var3= 'ODD: ';
    SET var4 = var3 || CHAR(counter);
  END IF;
until counter = 500
END REPEAT xLoop;
END;
```

Figure 18: Stored-procedure source code (scenario 2.a)

Test results and analysis (scenario 2.a)

According to the database-monitor trace, of the eight assignment and condition statements (for each loop iteration), four are implemented with Expression Evaluator. Similar to scenario 1, the effect of the ODP reuse within a loop means that only the first invocation of the stored procedure sees the biggest improvement. In fact, the performance results are quite consistent with those of scenario 1 (see Table 1).

Execution (within same job or connection)	Average i5/OS V5R4 run-time improvement
1	78.52 percent
2	19.18 percent
3	16.68 percent
4	13.44 percent

Table 12: Results for scenario-2.a testing

Scenario 2.b: Loop with conditions and assignments

For this scenario, the procedure contains a loop construct with 500 iterations. Within each loop iteration, five sets of IF-THEN-ELSE statements and 13 SET statements are run. The purpose of this scenario is to measure the impact of Expression Evaluator within a tight-looping construct with both conditions and assignment statements.

Stored-procedure source code (scenario 2.b)

Statements shown in blue are implemented with Expression Evaluator in i5/OS V5R4 (see Figure 19).

```
CREATE PROCEDURE Loop500_Conditions_and_Assignments_2()
LANGUAGE SQL
BEGIN

DECLARE var1 CHAR(3);
DECLARE var2 CHAR(1);
DECLARE var3 CHAR(5) CCSID 65535;
DECLARE var4 CHAR(10) CCSID 65535;
DECLARE counter INTEGER;

SET counter = 0;
SET var3 = 'EVEN';
xLoop:
REPEAT
  SET counter = counter + 1;
  SET var1 = 'A  ';
  SET var2 = TRIM(var1);

  IF MOD(Counter,2) + 10 = 10
  THEN
    SET var3= 'EVEN: ';
    SET var4 = var3 || CHAR(counter);
  ELSE
    SET var3= 'ODD: ';
    SET var4 = var3 || CHAR(counter);
  END IF;

  IF MOD(Counter,2) + 1 = 1
  THEN
    SET var3= 'EVEN: ';
    SET var4 = var3 || CHAR(counter);
  ELSE
    SET var3= 'ODD: ';
    SET var4 = var3 || CHAR(counter);
  END IF;

  IF MOD(Counter,2)  + 2 = 2
  THEN
    SET var3= 'EVEN: ';
    SET var4 = var3 || CHAR(counter);
  ELSE
    SET var3= 'ODD: ';
    SET var4 = var3 || CHAR(counter);
  END IF;

  IF MOD(Counter,2) + 3 = 3
  THEN
    SET var3= 'EVEN: ';
    SET var4 = var3 || CHAR(counter);
  ELSE
    SET var3= 'ODD: ';
    SET var4 = var3 || CHAR(counter);
  END IF;
  IF MOD(Counter,2) + 4 = 4
  THEN
    SET var3= 'EVEN: ';
    SET var4 = var3 || CHAR(counter);
  ELSE
    SET var3= 'ODD: ';
    SET var4 = var3 || CHAR(counter);
  END IF;
UNTIL counter = 500
END REPEAT xLoop;
END;
```

Figure 19: Stored-procedure source code (scenario 2.b)

Test results and analysis (scenario 2.b)

Of the 28 assignment and condition statements (for each loop iteration), 16 are implemented with Expression Evaluator. Table 13 shows the results of this scenario testing.

Execution (within same job or connection)	Average i5/OS V5R4 runtime improvement
1	70.01 percent
2	19.18 percent
3	1.09 percent
4	0.32 percent

Table 13: Results for scenario-2.b testing

Scenario 3: Multiple stand-alone statements

In the scenario, the procedure has a total of 305 individual, stand-alone statements with expressions. There are 100 sets of IF-THEN-ELSE statements and 205 SET statements. The goal of this scenario is to determine the influence on performance of Expression Evaluator when multiple statements are not within a looping construct. On i5/OS V5R3, each statement that implements the QSQPTABL creates on ODP. Consequently, these statements do not benefit from ODP reuse within a loop.

Stored-procedure source code (scenario 3)

Statements shown in blue are implemented with Expression Evaluator in i5/OS V5R4 (see Figure 20).

```
CREATE PROCEDURE stand-alone_statements_100()

LANGUAGE SQL
BEGIN
DECLARE v1 CHAR(1);
DECLARE v2 CHAR(1);
DECLARE v3 CHAR(7) CCSID 65535;
DECLARE v4 CHAR(6) CCSID 65535;
DECLARE v5 CHAR(10) CCSID 65535;
DECLARE counter INTEGER;

SET counter = 0;
SET v1 = 'A';
SET v2 = TRIM(v1);
SET v3 = 'EVEN: ';
SET v4 = 'ODD: ';
SET counter = counter + 1;
IF MOD(Counter,2) = 0
THEN
   SET v5 = v3 || CHAR(counter);
ELSE
   SET v5 = v4 || CHAR(counter);
END IF;

SET counter = counter + 1;
IF MOD(Counter,2) = 0
THEN
   SET v5 = v3 || CHAR(counter);
ELSE
   SET v5 = v4 || CHAR(counter);
END IF;
END;

--The above block of code is simply repeated 98 more times…it has been omitted to save space
```

Figure 20: Stored-procedure source code (scenario 3)

Test results and analysis (scenario 3)

When comparing the results of scenario 3 with the previous two scenarios, one thing that jumps out is the noticeable improvement in performance after the first stored-procedure invocation within the job. The reason for this disparity can be traced to the overhead of creating, maintaining and managing ODPs and the system resources that they require. Consider that each i5/OS V5R3 statement that implements the QSQPTABL dummy-table method creates an ODP. Recall that, in a stand-alone statement environment, each such statement incurs this overhead (305 statements in this scenario). In a looping-construct environment, this overhead is less-pronounced because the statements are within a loop and only incur the overhead during the first and second iterations of the loop (see Table 14).

Execution (within same job or connection)	Average i5/OS V5R4 runtime improvement
1	51.49 percent
2	90.11 percent
3	90.87 percent
4	90.82 percent

Table 14: Results for scenario 1 testing

Scenario 4: UDF implementation

The last scenario implements a user-defined function (UDF) that is started once for each matching row of an SQL SELECT statement. Two parameters are passed in the UDF. If either is NULL or ‘NIL’ or if the values are equal, an integer value of **1** is returned.

Stored procedure source code (scenario 4)

Statements shown in blue are implemented with Expression Evaluator in i5/OS V5R4 (see Figure 21).

```
CREATE FUNCTION CHECK_NULL (
 V_SOURCE_VAL VARCHAR(100) ,
 V_TARGET_VAL VARCHAR(100) )
 RETURNS INTEGER

 LANGUAGE SQL
 SPECIFIC Q_NULL_TEST
 NOT DETERMINISTIC
 READS SQL DATA
 CALLED ON NULL INPUT
 NO EXTERNAL ACTION
 NOT FENCED
 BEGIN ATOMIC

   IF ( ( ( ( V_SOURCE_VAL = 'NIL' ) OR ( V_SOURCE_VAL IS NULL ) ) AND ( ( V_TARGET_VAL = 'NIL' )
       OR ( V_TARGET_VAL IS NULL ) ) ) OR ( V_SOURCE_VAL = V_TARGET_VAL ) ) THEN
    RETURN 1 ;
   END IF ;

   RETURN 0 ;
  END  ;
```

Figure 21: Stored-procedure source code (scenario 4)

Test results and analysis (scenario 4)

The following statement is run to start and test the UDF:

```
SELECT COUNT(*) FROM CUSTOMERS WHERE CHECK_NULL(ADDRESS,TERRITORY) = 1;
```

When the above statement is run, CUSTOMERS table has 150 000 rows and 6100 matching rows are returned (see Table 15).

Execution (within same job or connection)	**Average i5/OS V5R4 run-time improvement**
1	79.33 percent
2	78.95 percent
3	79.17 percent
4	79.28 percent

Table 15: Results for scenario-4 testing

Appendix B: C code generation conditions

This section provides conditions in which C code is generated to perform declaration, assignment and comparison statements in the SQL procedural language. (IBM reserves the right to change these conditions.)

Note: The C code generation method is never used for the first statement within an SQL Trigger or Function. In addition, C code generation does not occur for any variable declared with the UTF-8 CCSID (1208), UTF-16 CCSID (1200), or statements referencing hex literals (for example, SET v1 = X’C1F1F2’).

Declarations and assignment statements

For declaration and assignment statements, the following list describes the conditions that must exist for C Code generation to occur

- For statements with the formats **DECLARE V1 default <numeric value>** or **SET V1 = <numeric value>**, C code is generated if <value> is:
    - NULL.
    - 0 for all numeric types.
    - Integer value and V1 is integer or bigint or V1 is smallint and value is not truncated.
    - Decimal value and V1 is decimal or numeric with length/scale >= length/scale of constant.
    - Bigint value and V1 is bigint.
- For statements with the formats **DECLARE V1 default <string value>** or **SET V1 = <string value>,** C code is generated is:
    - When <value> is NULL
    - When <value> is '' and V1 is char or varchar.
    - When V1 is char and the variable CCSID is not UTF-8 and the source statement CCSID is 65535 and the string length <= 256 and less than equal to the length V1.
    - Both C code and an SQL SET statement are generated when V1 is varchar and the length of V1 is greater than or equal to the length of the string and the string length <= 256 and the CCSID of V1 is not specified At runtime, the generated C code will be used when job CCSID matches the source statement CCSID - if not, the SQL SET statement is invoked.

- For statements with the format **SET V1 = V2**
    - C code is generated if V1 and V2 are the same numeric type.
      Note: For decimal and numeric variables, the length (or scale) of the target variable must be greater than, or equal to, the source variable. For float, the length of the target variable must be greater than, or equal to, the source variable
    - C code is generated if both V1 and V2 are either date, time, or timestamp
    - C code is generated if all of the following conditions are true for V1 and V2:
        - Both V1 & V2 are char or both V1 & V2 are varchar or both V1 & V2 are graphic or both V1 & V2 are vargraphic
        - The CCSIDs are the same

        - The length of V1 is greater than, or equal to, the length of the V2
    - When V1 and V2 are different character data types and the CCSIDs are the same, C code can be generated in the following conditions when V1 & V2 are the same and the length of V1 is greater than or equal to the length of V2.:
        - If V1 is varchar and V2 is char.
        - If V1 is char and V2 is varchar.
        - If V1 is graphic and V2 is vargraphic.
        - If V1 is vargraphic and V2 is graphic.

- For statements with the format **SET V1 = V1 + <constant>** or **SET V1 = V1 - <constant>**
    - C code is generated to add constant literal if V1 is an Integer, SmallInt, or BigInt. Prior to the IBM i 7.1 release, C code can only be generated when the constant literal value is 1 and V1 is an integer.

- For statements with the format **SET V1 = V1 || <constant>**
    - C code is generated if V1 is varchar, if length of <constant> is <= 256 and CCSID of V1 is not UTF-8.
      Note: If the CCSID of v1 is 65535, only C code is generated. If the CCSID of v1 is not specified and mixed data is not specified and target release of i5/OS is V5R4 or later, C code is generated that checks the job CCSID when the routine is called. If the job CCSID is 65535, C code is run, otherwise SQL is called to implement the task using one of the other methods.

- For statements with the format SET V1 = SUBSTRING (V2, 1, constant)
    - C code is generated when V1 and V2 are graphic or vargraphic if all of the following conditions are true:
        - The CCSIDs for V1 and V2 are the same
        - The constant value is a non negative value which is less than or equal to the length of V2 and less than or equal to the length of V1

- For GET DIAGNOSTICS statements, C code is generated under the following conditions:
    - The statement only retrieves either the DB2_RETURN_STATUS or ROW_COUNT diagnostics and the receiving variable is an integer or bigint

o

Comparison statements

For comparison statements, the following list describes the conditions that must exist for C Code generation to occur. If the comparison contains an AND or an OR logical operator, the C code generation method cannot be used.

- For comparison statements with the format **IF V1 compare <numeric value>**
  - C code is generated if V1 is smallint, integer, bigint, decimal or numeric and <value> is integer, bigint or decimal.
- For comparison statements with the format **IF V1 compare <string value>** and no sort sequence specified (sort sequence setting is set to *HEX and not *JOBRUN).

  - If V1 is char or varchar
    - C code is generated when all of the following conditions are true:
      - Lengths of V1 and <value> are the same and <= 256
      - The source CCSID is 65535 and no CCSID is specified for V1
      - The comparison operator is =, >, < or <>
    - C and SQL code are generated all of the following conditions are true:
      - Lengths of V1 and <value> are the same and<= 256
      - The source CCSID is not 65535 and no CCSID is specified for V1
      - The comparison operator is =, >, < or <>

    Note: The C code is run at runtime whenever the runtime job CCSID matches the source statement CCSID. Otherwise, the generated SQL is executed.

- For comparison statements using the format **IF V1 compare <string value>** with a unique weight sort sequence specified (assuming sort sequence does not equal *JOBRUN)
  - If V1 is char or varchar, then
    - C code is generated when all of the following conditions are true:
      - Lengths of V1 and <value> are the same and <= 256
      - The source CCSID is 65535 and no CCSID is specified for V1
      - The comparison operator is = or <>
    - C and SQL code are generated when all of the following conditions are true:
      - Lengths of V1 and <value> are the same and <= 256
      - The source CCSID is not 65535 and no CCSID is specified for V1
      - The comparison operator is = or <>

    Note: The C code is run at runtime whenever the runtime job CCSID matches the source statement CCSID. Otherwise, the generated SQL is executed.

- For comparison statements with the format **IF V1 comp V2**
  - C code is generated if V1 and V2 are smallint, integer, bigint, decimal or numeric
  - C code is generated when V1 and V2 are char if all of the following conditions are true:
    - Lengths of V1 and V2 are same
    - The CCSIDs for V1 & V2 are the same
    - No sort sequence is specified (that is, *HEX), cannot be *JOBRUN
    - The comparison operator is =, >, < or <>

- For comparison statements with the following format:

  IF V1 IN (character-string constant, …) THEN …
  IF V1 NOT IN (character-string constant, …) THEN
  IF V1 IN (Unicode graphic-string constant, …) THEN …
  IF V1 NOT IN (Unicode graphic-string constant, …) THEN
  - C code is generated when V1 is graphic or vargraphic if all of the following conditions are true:
    - The CCSIDs for V1 is 13488
    - The constant string is 256 characters or less in length
    - The IN or NOT IN predicate contains 3 constants or less

Resources

These Web sites provide useful references to supplement the information contained in this document:

- IBM i Documentation
  ibm.com/docs/en/i
- Db2 for i online manuals
  ibm.biz**/**Db2iBooks
- Db2 for i Blog
  Db2Ibmi.blogspot.com
- SQL Performance Assessment and Enablement
  ibm.biz/Db2iExpertLabs
- SQL Query Engine (SQE)
  ibm.com/support/knowledgecenter/en/ssw_ibm_i_74/rzajq/queryoptimize.htm
- IBM i Access Client Solutions
  ibm.com/support/pages/ibm-i-access-client-solutions
- Db2 for i Technology Updates and Group PTFs
  ibm.com/ibmi/techupdates/Db2
- Db2 for i Redbooks
  ibm.biz/Db2iRedbooks

About the Authors

Kent Milligan is a Senior Db2 for i Consultant in the IBM Technology Expert Labs. Kent has over 25 years of experience as a Db2 for IBM i consultant and developer working out of the IBM Rochester lab. Prior to re-joining the DB2 for i Expert Labs practice in 2020, Kent spent 5 years working on healthcare solutions powered by IBM Watson technologies. Kent is a sought-after speaker and author on Db2 for i & SQL topics.

Acknowledgements

Thanks to the following people who reviewed and contributed to this paper:

- Scott Hanson
- Jarek Miszczyzk
- Gene Cobb
- Rob Bestgen
- John Broich
- Christian Griere
- Shantan Kethireddy
- Kathy Passe

Trademarks and special notices