Out in Left Field

Volume IV, Issue VII

.NET and iSeries Application Architecture

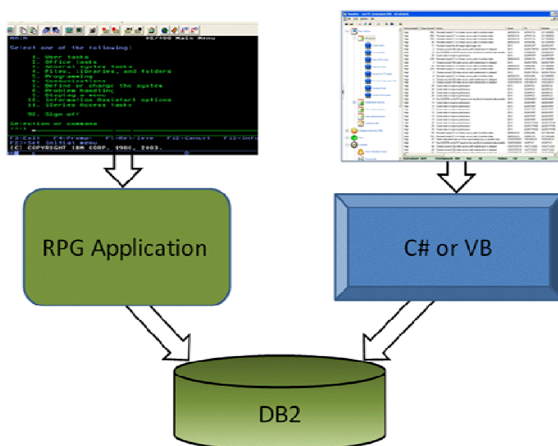
As with all integrated development environments (IDEs), Microsoft's .NET platform provides a wide variety of techniques to implement new features. The rich development environment and language choices make the .NET platform an attractive choice for many development shops – particularly with the plentiful supply of young programmers who have strong Microsoft knowledge.

With the technical flexibility and the availability of skill, many iSeries shops are using .NET as the platform to extend their existing applications. The flexibility of .NET allows a developer to design and build their application using a broad set of techniques. Typically however, there is a bias toward a particular design approach. No matter what design method is used, nearly all teams would like to achieve one that minimizes long-term maintenance costs, can be done in the shortest possible time, performs well, and of course, appeals to their end-users.

The purpose of this article is to explore different techniques and analyze the strengths and weaknesses of each method.

Client-heavy Architecture

One plan of attack is to take a client-centered perspective. [Note: for the purposes of this article, I am ignoring web development but many of the concepts in this section apply to this methodology as well.] In this pattern, the objective is to minimize the impact to the code of an existing application and either rewrite existing function on the client or only implement new features on the client. Often this approach is taken because there is great resistance to change an existing application. The reluctance is very natural because no one wants to break a working application or invest a lot of energy to understand code that may have been written 20 years ago.
At a high level, the "application" looks similar to the following:



In this model, a legacy 5250 application remains unchanged while C# or Visual Basic extensions are written to enhance the functionality of a core business process. Generally the modernized part of the application builds on the existing database and in all likelihood extends it as well.

If legacy code remains static but it has been decided that some of its functions need to be modernized, the only option is rewrite those functions. On the other hand, if the strategy is to only write **new** features in .NET, then the end-user may need to flip between a green screen and the client. Both of these approaches, while minimizing the need to deal with old code, are not ideal.
Besides forcing the end-user to work partially with a green screen and partly with a graphical interface, this approach has the serious disadvantage of maintainability. Let's say for example, that application enhancements require changes to the database.

Those changes may force changes to the existing RPG code (if the change is driven by the client) or force changes to the client (if the change is driven by the RPG code). Either way, maintenance is not only more complicated but it is more likely that errors will occur since two separate teams must stay in synch.

One way some development teams work around this is to replicate data so each piece of the application works with its own tables. Needless to say, the hidden cost is the need to synchronize that data and avoid having the two data sets report different information.

(Continued on page 3)

Inside this issue:

.Net and iSeries Application Architecture	1, 3-4
CTO Memo	2
Centerfield Training	2
Enhancing SQL Triggers	5-7
The Unconstrained Primary Key	8
Finding Differences Between Two Libraries	8-10

Welcome to our May newsletter. We hope this newsletter finds you enjoying the warm spring weather.

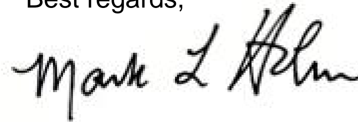
I'd like to start off by thanking Dan Cruikshank for his presentation at the recent Centerfield sponsored webinar entitled "The Unconstrained Primary Key." Based on the attendance at this webinar and requests for the video and presentation materials, it was a very popular topic. As the iSeries community takes advantage of SQL to define database tables and moves away from DDS, presentations like this provide very valuable guidance and advice. The webinar video can be found on Centerfield's website on the Events tab.

The articles we have this month will help grow your knowledge of DB2 and SQL. As usual, Elvis has a couple of excellent articles on clever and powerful uses for SQL. He combines SQL with CL programming and IBM APIs to solve some very practical problems. You'll enjoy them I'm sure.

As your shop utilizes SQL and advanced DB2 features, please feel free to contact Centerfield Technology. We are here to help you get the most out of the iSeries and take advantage of the stability of the platform in combination with the latest SQL and DB2 technology. We are the team to talk to if you are faced with SQL performance challenges.

Have a wonderful spring wherever you are!

Best regards,



Mark Holm

Centerfield's Upcoming Training Opportunity

HomeRun Training—Part II

Friday, May 8, 2009
10:00 AM - 11:30 PM CDT

The ADVANCED Course includes:

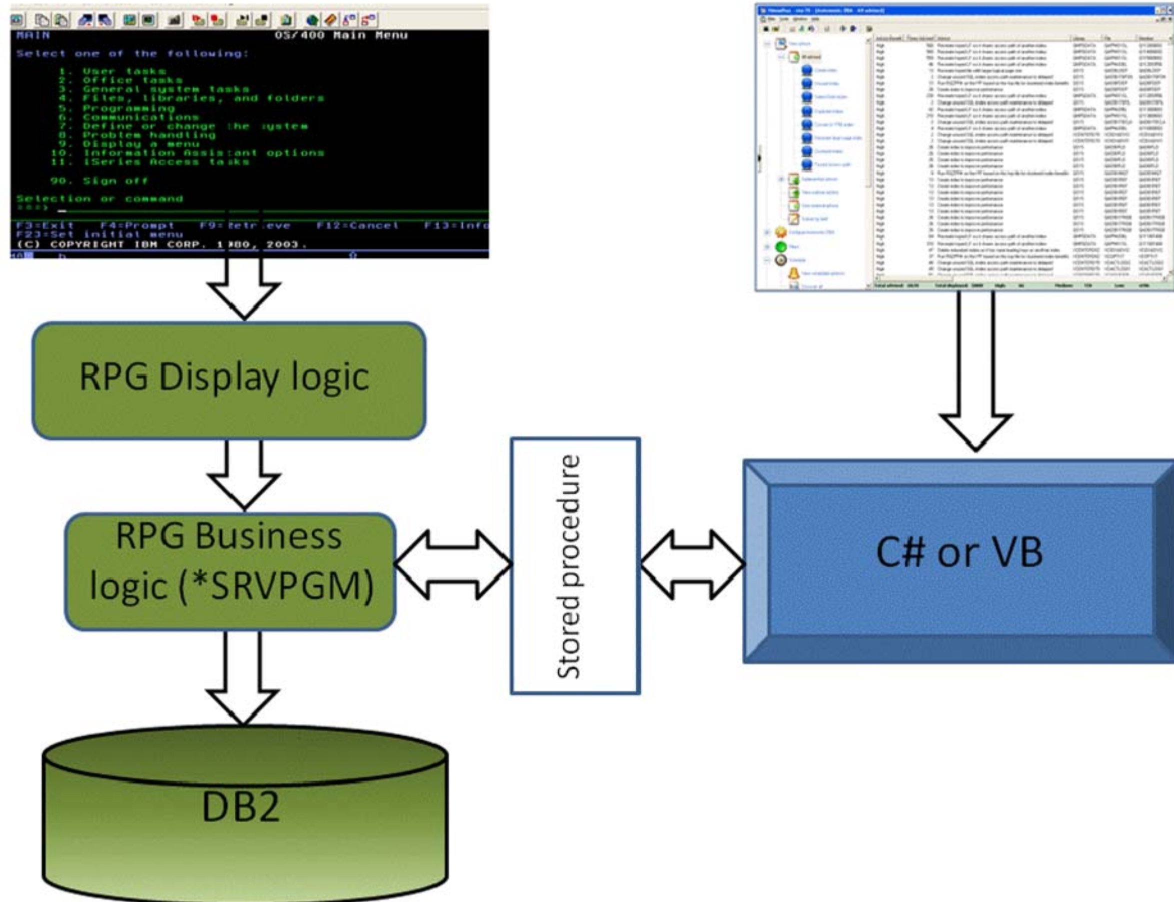
- Insure/INDEX (AutoDBA)
- ANALYSIS
- RESOURCES
- SECURITY
- MONITOR

Sign up TODAY!

(.Net and iSeries Application Architecture—Continued from page 1)

Host-centric Architecture

Another design model is to take more of a host-centric approach. The philosophy behind this approach is to leverage the existing business logic embedded in RPG code as much as possible. Essentially this means developing a set of common functions used by the 5250 portions of the application as well as the .NET extensions.

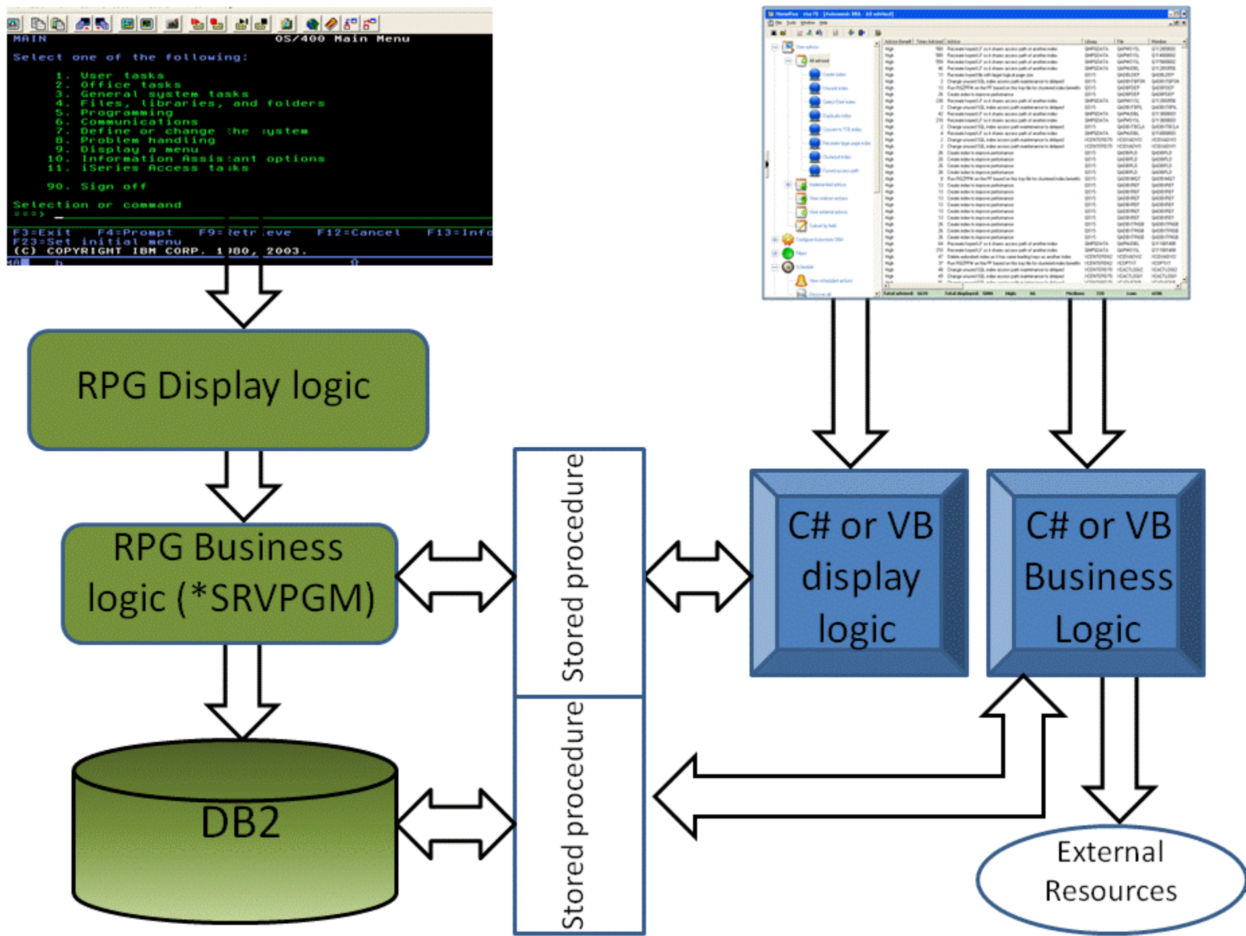


To implement this architecture, the display logic in the RPG application must be separated from the core business logic. With the intelligent use of ILE service programs this can be done in a very modular and organized way. Once the core logic is split out, SQL or RPG stored procedures can be written as wrappers to the core logic.
This architecture has several, very important advantages:

1. Breaking the presentation logic from business functions makes the RPG portion of the application easier to maintain and enhance.
2. Well-tested and trusted business rules do not have to be re-written and that saves not only time and resources, but will almost certainly reduce bugs.
3. Changes to either the business logic or the database will result in less impact to the client-side portion of the application.
4. Performance can be much better because there are fewer trips from the client to the server to read and write data.

Hybrid Architecture

One of the key benefits of the .NET platform is the ease of which applications can take advantage of code and services written by others. Furthermore, it may be necessary to integrate a legacy application with a stand-alone business function on a platform other than an iSeries. To leverage pre-written services or third-party software is best implemented using a hybrid architecture as depicted here.



The fundamental idea is to keep the benefits of the host-centric architecture while leveraging the power of the .NET platform to add function to an application very quickly. Even in this environment, it is best to use stored procedures to access DB2 data for the reasons described above. In this scenario, the .NET client becomes the “master integration point” for the legacy components, web services, and other client-based applications.

Summary

We’ve discussed several different architectures using an iSeries application and the .NET platform. While there are no absolutes when building a robust business application, it makes sense to thoroughly look at the tradeoffs (both short and long term) before deciding the best approach for your use of these cooperative technologies.

Enhancing SQL Triggers

Enhancing SQL triggers

Native database trigger support was introduced in the V3R1 release of the OS/400 (1994). Since its introduction, trigger usage has exploded in popularity. We use it for ETL processing, auditing, integration with other platforms and many other wonderful functions limited only by our imagination. The verdict is in and triggers are one of those 'killer functions' that make our lives easier and IT more productive. Native triggers can be written in any high level language, but of course the most popular is RPG.

SQL trigger support was introduced in V5R1 (2001), so while it is a relative newcomer, it is a mature feature of the IBM i operating system. These types of triggers are written in SQL Stored Procedure Language (SPL) instead of COBOL or RPG.

SQL triggers are starting to appear on the System i more often for a couple of reasons. Since on other database platforms SQL is the only way to interface with the database, it shouldn't come as a surprise that non-i DBAs (Database Administrators) and developers write SQL triggers. If they are lucky enough that their applications achieve wide commercial success, they may eventually migrate their product to the System i. Needless to say, SQL triggers come along with it.

In another scenario, perhaps you've hired a person experienced on the non-i databases and they just feel more comfortable writing in SQL than using one of the HLL languages offered on the System i.

Inevitably, SQL trigger developers will be tapped by a System i colleague to enhance their SQL trigger to include additional diagnostic/audit information easily available to all native triggers. That information might include job name, user profile accessing the table and a program that fired the trigger. It could also include other diagnostic information like an IP address or a port of the remote connection.
Unbeknownst to you, you've just presented your SQL developer with quite a challenge. In a native RPG trigger, we have most of this information readily available in the PSDS (Program Status Data Structure). This data is not as easily accessible in SPL. One obvious solution is to switch to native trigger support, but that may not be an option if portability needs to be preserved.

This article is intended to help an SQL trigger developer tap CL (Control Language) UDF (User Defined Function) support to add diagnostic information like job name, user profile and program name to the audit trail. Furthermore, the CL code will demonstrate how invaluable V5R4 CL enhancements are when we need to call a complex API like QWVRCSTK (Retrieve Call Stack information).

CL code for UDFs

Before we can create an SQL trigger, the UDFs we will use within it need to be registered with the DB2 for i SQL catalogues. It's a good practice to have the target service program object pre-created before we register the UDFs so let's build the base CL objects.
We will create a service program binding two distinct CL modules in it. The first CL module will map to the RetrieveJobName() UDF and second will map to the RetrieveFiringTriggerProgram () UDF.
One important note pertaining to the CL code is that I ignore all of the non-essential parameters involved with the SQL parameter style. In other words, even though DB2 passes arguments like null indicators, SQLSTATE, function name, special name, message text and output value's null indicator, I do not receive or use them in my CL programs for brevity purposes. If this was a commercial application, I would want to accept and handle those arguments appropriately. Robust error handling is also left out to keep the code simple.

```
/* Start CL source for RTVJOBNAME – RetrieveJobName() UDF */

PGM (&qualjob)
  dcl &qualjob *char 28  /* output argument, fully qualified job name */

  /* local CL variables */
  dcl &job  *char 10
  dcl &user *char 10
  dcl &nbr  *char  6

  RTVJOBA JOB(&JOB) USER(&USER) NBR(&NBR)

  chgvar  &qualjob (&nbr *cat '/' *cat &user *tcat '/' *tcat &job)

ENDPGM

/* end CL source for RTVJOBNAME – RetrieveJobName() UDF */
```

What follows is the CL source to retrieve the trigger-firing program name. While CL is not the best language for this type of (complex) logic, I wanted to illustrate how V5R4 CL enhancements ease list API processing.

```
/* Start CL source for RTVTRGPGM – RetrieveFiringTriggerProgram() UDF */

PGM (&trigpgm &triglib &qualpgm &ind1 &ind2)

dcl &trigpgm *char 10  /* trigger program name    */
dcl &triglib *char 10  /* trigger program library */
dcl &qualpgm *char 21  /* output argument, fully qualified program name */
dcl &ind1 *int 2
dcl &ind2 *int 2

/* local CL variables */
dcl &receiver  *char 32767
  dcl &offset      *int stg(*defined) defvar(&receiver 13)
  dcl &numentry    *int stg(*defined) defvar(&receiver 17)
  dcl &callStkPtr *ptr address(&receiver 1)
  dcl &callStkLen *int stg(*based) basptr(&callStkPtr)
  dcl &pgm         *char 10
  dcl &pgmlib      *char 10
  dcl &pgmoffset *int 4 value(0)
dcl &rcvlen    *int  value(32767)
dcl &fmt       *char 8 'CSTK0100'
dcl &jobident  *char 56
  dcl &jobname    *char 10 stg(*defined) defvar(&jobident  1)
  dcl &user       *char 10 stg(*defined) defvar(&jobident 11)
  dcl &jobnbr     *char  6 stg(*defined) defvar(&jobident 21)
  dcl &intjobid  *char 16 stg(*defined) defvar(&jobident 27)
  dcl &reserved  *char  2 stg(*defined) defvar(&jobident 43)
  dcl &threadind *int     stg(*defined) defvar(&jobident 45)
  dcl &threadid  *char  8 stg(*defined) defvar(&jobident 49)
dcl &jobfmt    *char 8 'JIDF0100'
dcl &errorCode *char 8 x'0000000000000000'

dcl &limit      *int /* loop control variable */

/* local variables for CL service program name retrieval */
dcl &matPgmData *char 80
  dcl &dataSize   *int     stg(*defined) defvar(&matPgmData  1)
  dcl &bytesAvail *int     stg(*defined) defvar(&matPgmData  5)
  dcl &pgmFormat  *int     stg(*defined) defvar(&matPgmData  9)
  dcl &reserved2  *int     stg(*defined) defvar(&matPgmData 13)
  dcl &thispgmlib *char 10 stg(*defined) defvar(&matPgmData 19)
  dcl &thispgmnam *char 10 stg(*defined) defvar(&matPgmData 51)

/* initialize input arguments for the _MATPGMNM MI built-in call */
chgvar &dataSize   80
chgvar &bytesAvail 80
chgvar &pgmFormat   0
chgvar &reserved2   0

/* get qualified service program name, so we can omit it from consideration */
CALLPRC PRC('_MATPGMNM') PARM(&matPgmData)

/* initialize QWVRCSTK input arguments */
chgvar &jobname   '*'
chgvar &reserved  x'0000'
chgvar &threadind 1
chgvar &threadid  x'0000000000000000'

call QWVRCSTK parm(&receiver &rcvlen &fmt &jobident &jobfmt &errorCode)

/* walk the call stack list and get the firing trigger program name */
doFor VAR(&LIMIT) FROM(1) TO(&NUMENTRY)
  chgvar &callstkptr %address(&receiver)
  chgvar %offset(&callstkptr) (%offset(&callstkptr) + &offset)
  chgvar &pgmoffset (&offset + 25)
  chgvar &pgm %sst(&receiver &pgmoffset 10)
  chgvar &pgmoffset (&pgmoffset + 10)
  chgvar &pgmlib %sst(&receiver &pgmoffset 10)

  /* omit QSYS programs, this service program, and SQL trigger program name */
  if (&pgmlib *NE 'QSYS' *AND +
      *NOT (&pgm = &thispgmnam *AND &pgmlib = &thispgmlib) *AND +
      *NOT (&pgm = &trigpgm *AND &pgmlib = &triglib)) +
  then(do) /* firing trigger program found */
    chgvar &qualpgm  (&pgmlib *tcat '/' *tcat &pgm)
    return
  enddo
  chgvar &offset (&offset + &callstklen)
enddo

/* entire call stack was QSYS programs, so let's special case this */
chgvar &qualPgm '*IBM_ONLY'

ENDPGM

/* Start CL source for RTVTRGPGM – RetrieveFiringTriggerProgram() UDF */
```

Let’s create the service program object, which will serve as a target for the UDFs we will register:

```
CRTCLMOD MODULE(QGPL/RTVJOBNAME) SRCFILE(QGPL/QCLSRC)
CRTCLMOD MODULE(QGPL/RTVTRGPGM) SRCFILE(QGPL/QCLSRC)
CRTSRVPGM SRVPGM(QGPL/UDFS) MODULE(QGPL/RTVJOBNAME QGPL/RTVTRGPGM) EXPORT
(*ALL)
```

Let’s register the UDFs to be used in the SQL trigger code:

```
CREATE FUNCTION QGPL/RetrieveJobName()
RETURNS CHAR(28)
EXTERNAL NAME 'QGPL/UDFS(RTVJOBNAME)'
LANGUAGE CL
PARAMETER STYLE SQL
DETERMINISTIC
NO SQL
RETURNS NULL ON NULL INPUT
NO EXTERNAL ACTION
DISALLOW PARALLEL
NO SCRATCHPAD
NOT FENCED

CREATE FUNCTION QGPL/RetrieveFiringTriggerProgram
(
  TriggerProgramName CHAR(10),
  TriggerProgramLibrary CHAR(10)
)
RETURNS CHAR(21)
EXTERNAL NAME 'QGPL/UDFS(RTVTRGPGM)'
LANGUAGE CL
PARAMETER STYLE SQL
```

SQL set-up

Let's create a table that has some application data, but more importantly includes columns that will contain the additional diagnostic information.

```
CREATE TABLE QGPL/ApplicationData
 (FIRST_NAME    CHAR (20),
  LAST_NAME     CHAR (20),
  ADDRESS       CHAR (50),
  AUDIT_JOB     CHAR (28),
  AUDIT_PROFILE CHAR (10),
  AUDIT_PROGRAM CHAR (21))
```

Next, let's create a simple *BEFORE INSERT SQL trigger that sets the audit columns correctly.
I need to point out couple of important notes relating to the SQL code within the trigger:

1) Even though I retrieve the short (system) trigger program name in the trigger code, for performance reasons, I recommend you hardcode the value in production code.
2) For the same reason I decided to leverage the SQL special register USER for the Current Profile value instead of coding yet another UDF calling RTVJOBA.

```
CREATE TRIGGER qgpl/AddCustomer
  BEFORE INSERT ON qgpl/ApplicationData
  REFERENCING NEW ROW AS newRow
  FOR EACH ROW MODE DB2ROW
BEGIN
  DECLARE TriggerProgramName    CHAR(10);
  DECLARE TriggerProgramLibrary CHAR(10);

  SELECT TRIGGER_PROGRAM_NAME,TRIGGER_PROGRAM_LIBRARY
  INTO TriggerProgramName,TriggerProgramLibrary
  FROM QSYS2/SYSTRIGGERS
  WHERE TRIGGER_SCHEMA = 'QGPL' AND TRIGGER_NAME = 'ADDCUSTOMER';

  SET (newRow.AUDIT_JOB,newRow.AUDIT_PROFILE,
  newRow.AUDIT_PROGRAM) =
      (RetrieveJobName(),
       USER,
       RetrieveFiringTriggerProgram(TriggerProgramName,
                            TriggerProgramLibrary));
END
```

Testing the INSERTs

For SQL trigger testing, I am going to use INSERT statements similar to the following:

```
INSERT INTO qgpl/ApplicationData (FIRST_NAME,LAST_NAME,ADDRESS)
VALUES('Elvis','Budimlic','3131 Superior Dr. NW, Suite C')
```

Here is a sample output from three different INSERTs I've performed:

FIRST_NAME	LAST_NAME	ADDRESS	AUDIT_PROFILE	AUDIT_PROGRAM	AUDIT_JOB
Elvis	Budimlic	3131 Superior Dr. NW, Suite C	*113570/ELVIS/ QPADEV0006*	*ELVIS*	ELVIS/MYINSPGM
Mark	*Holm*	*3131 Superior Dr. NW, Suite C*	*113570/MLH/ QPADEV0006*	*MLH*	*QSQL/QSQISE*
Aaron	*Hasley*	*3131 Superior Dr. NW, Suite C*	*113747/QUSER/ QZDASOINIT ELVIS*	*ELVIS*	**IBM_ONLY*

The first insert was done by an HLL program called in an interactive green-screen session. The program name is MYINSPGM and is reflected in the AUDIT_PROGRAM column. The remaining information is self-explanatory.

A second insert was done from an STRSQL (interactive SQL) session. Notice that the program name is QSQL/QSQISE as that is the first non-QSYS program in the call stack. In fact, if you've ever looked at your call stack in the command line, you'll notice that there are no 'user' entries in there, just IBM system and IBM product programs (STRSQL is part of the DB2 Query Manager and SQL Development Kit product). Thus, QSQL/QSQISE may be interpreted and logged as STRSQL instead. To accomplish that, you'd need to modify the RTVTRGPGM code I provided.

The third insert was done from the IBM Navigator's Run SQL Script facility. As you can see, all program names in the stack come from the QSYS library, so our code inserted a special value of *IBM_ONLY.

Conclusion

This article illustrates how to integrate SQL triggers, CL UDFs and handling of complex API lists leveraging V5R4 CL enhancements. Further, it illustrates retrieval of a firing trigger program name, which is a frequent requirement for all database triggers used for auditing purposes.

I hope you find one or more of these techniques useful in your work.

Finding Differences Between Two Libraries

I often talk to customers who have multiple 'environments' or 'instances' of a particular application on the same LPAR. This usually translates to several copies of the same database, just in different libraries. For example, you may have a **PROD**uction and a **TEST** library.

Inevitably, they want to compare the two libraries for differences. One can make this comparison as complex as they want, but a simple object check will often suffice. Following article offers one alternative for a quick 'sanity' check on the differences between two libraries.

EXCEPT versus EXCEPTION JOIN

In thinking how to perform a quick and simple check on the differences between two files that contain information about the libraries, the EXCEPT SQL keyword seemed like an obvious candidate:
http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/sqlp/rbafyexcept.htm

It is a variant on the LEFT EXCEPTION JOIN of which I have written about in our February 2008 newsletter: http://www.centerfieldtechnology.com/publications/archive/FebMar08.pdf

Unlike LEFT EXCEPTION JOIN, the EXCEPT method removes duplicates from the final result set (because of its implicit DISTINCT nature) and it combines two result sets on ALL projected columns rather than select ones.

Because of this behavior, I encountered problems trying to project additional columns like external program name for a trigger since it includes library names, making it distinct and eliminating it as a join candidate. Also, simulating a FULL EXCEPTION JOIN with it is more cumbersome than with the EXCEPTION JOIN clause.

Since the EXCEPT clause turned out to be overly restrictive, I decided to fall back to the trusted LEFT EXCEPTION JOIN.

Planning the compare

First of all, we need to make a decision about which objects to compare between the two libraries, so we can scope our task accordingly. Is it just data objects like tables, or should we compare program objects as well? What about triggers, UDFs, and stored procedures?
Once we're clear on the plan, we could create a script and run it via RUNSQLSTM or Run SQL Scripts tools.
For the purposes of this article, I will compare data objects (table/view/alias/PF/LF), UDFs, stored procedures, triggers and illustrate a rudimentary compare of all 'other', traditional objects (programs, commands, data queues, message queues, user spaces etc.).

(Continued on page 9)

Key Topics Include:

- Row filtering using inclusion and exclusion tables
- Pagination using the PK with Row_Numbers
- Column sorting for large result sets

If you missed out on his live presentation, you aren't out of luck!

His presentation along with his Question and Answers session can be viewed just by clicking the link below:

The Unconstrained Primary Key
(http://www.centerfieldtechnology.com/UPK.asp)

Dan has spent the last 19 years with IBM and has over 35 years IT experience. His areas of expertise include application and database performance troubleshooting and optimization. He has developed many IBM internal tools for extracting and analyzing system and database performance data. He has written extensively on application performance tuning and database reengineering (DDS to DDL). He is currently one of the IBM Rochester certified instructors for the IBM i Performance Tuning Workshop.

(Finding Differences Between Two Libraries—Continued from page 8)

Stored procedure approach

I decided to go with a stored procedure approach, rather than an SQL script. The main reason is that I can pass parameters in easily, as well as illustrate the concept by invoking it in the IBM Navigator's Run SQL Script facility.
Here is the stored procedure code you can use as a skeleton for flexing out your own, more elaborate compare:

```
CREATE PROCEDURE LIBRARY_DIFFERENCES(LIBRARY1 CHAR(10), LIBRARY2 CHAR(10))
RESULT SETS 4
LANGUAGE SQL
BEGIN
  DECLARE STMT VARCHAR(200);
  DECLARE STMT_LEN INTEGER;
  DECLARE vLIBRARY1 CHAR(10);
  DECLARE vLIBRARY2 CHAR(10);
  DECLARE TABLE_DIFF CURSOR FOR
    WITH
    LIB1 AS
      (SELECT *
       FROM QSYS2/SYSTABLES
       WHERE FILE_TYPE = 'D' AND SYSTEM_TABLE_SCHEMA = vLIBRARY1),
    LIB2 AS
      (SELECT *
       FROM QSYS2/SYSTABLES
       WHERE FILE_TYPE = 'D' AND SYSTEM_TABLE_SCHEMA = vLIBRARY2)
    SELECT 'Table Differences' AS Table_Differences, LIB1.*
    FROM LIB1 LEFT EXCEPTION JOIN LIB2 USING (SYSTEM_TABLE_NAME,TABLE_TYPE);
  DECLARE TRIGGER_DIFF CURSOR FOR
    WITH
    LIB1 AS
      (SELECT *
       FROM QSYS2/SYSTRIGGERS
       WHERE TRIGGER_SCHEMA = vLIBRARY1),
    LIB2 AS
      (SELECT *
       FROM QSYS2/SYSTRIGGERS
       WHERE TRIGGER_SCHEMA = vLIBRARY2)
    SELECT 'Trigger Differences' AS Trigger_Differences, LIB1.*
    FROM LIB1 LEFT EXCEPTION JOIN LIB2 USING(TRIGGER_NAME,TRIGGER_PROGRAM_NAME);
  DECLARE ROUTINE_DIFF CURSOR FOR
    WITH
    LIB1 AS
      (SELECT *
       FROM QSYS2/SYSROUTINES
       WHERE ROUTINE_SCHEMA = vLIBRARY1),
    LIB2 AS
      (SELECT *
       FROM QSYS2/SYSROUTINES
       WHERE ROUTINE_SCHEMA = vLIBRARY2)
    SELECT 'Routine Differences' AS Routine_Differences, LIB1.*
    FROM LIB1 LEFT EXCEPTION JOIN LIB2 USING(ROUTINE_NAME,ROUTINE_TYPE);
  DECLARE OBJECT_DIFF CURSOR FOR
    SELECT 'Object Differences' AS Object_Differences, A.*
    FROM QTEMP/OUTFILE1 A LEFT EXCEPTION JOIN
         QTEMP/OUTFILE2 B USING(ODOBNM,ODOBTP,ODOBAT);


  SET (vLIBRARY1,vLIBRARY2) = (UPPER(LIBRARY1),UPPER(LIBRARY2));


  SET STMT = 'DSPOBJD OBJ(' CONCAT TRIM(vLIBRARY1) CONCAT
          '/*ALL) OBJTYPE(*ALL) OUTPUT(*OUTFILE) OUTFILE(QTEMP/OUTFILE1)';
  SET STMT_LEN = LENGTH(STMT);
  CALL QCMDEXC(STMT,STMT_LEN);
  SET STMT = 'DSPOBJD OBJ(' CONCAT TRIM(vLIBRARY2) CONCAT
          '/*ALL) OBJTYPE(*ALL) OUTPUT(*OUTFILE) OUTFILE(QTEMP/OUTFILE2)';
  SET STMT_LEN = LENGTH(STMT);
  CALL QCMDEXC(STMT,STMT_LEN);


  OPEN TABLE_DIFF;
  OPEN TRIGGER_DIFF;
  OPEN ROUTINE_DIFF;
  OPEN OBJECT_DIFF;
END;
```

To test the stored procedure just CALL it in Run SQL Scripts as:

```
CALL LIBRARY_DIFFERENCES('library1','library2');
```

Notice that I return 4 distinct result sets from this stored procedure. This is where Run SQL Scripts tool comes in really handy, as it'll by default display all four result sets within their own windows. If you'd like them in separate windows, you can click on Options-> Display Results in a Separate Window.

To Unionize or not

Instead of having 4 separate result sets, I could have taken a more traditional approach of providing a single result set. This would involve a UNION ALL of 4 distinct SQL statements. I would have to ensure that the columns projected from all 4 statements match not only in count but also in data type. If all you care about are simple attributes like object name and type, UNION ALL would make sense and be easy to accomplish.

Since I don't know what all readers will be interested in, I elected to stick with 4 different results sets and provide all of the available data.

Full Exception Join consideration

The code in the stored procedure will show objects in LIBRARY1 that are not in LIBRARY2. What if you also want to include objects in LIBRARY2 that are not in LIBRARY1? In SQL vernacular, this result set is called a FULL EXCEPTION JOIN. Presently, there is no explicit SQL keyword for it on DB2 for i, but it is very easy to simulate by executing a UNION ALL between the existing LEFT EXCEPTION JOIN and a RIGHT EXCEPTION JOIN.
Here is a modified example of the OBJECT_DIFF cursor:

```
DECLARE OBJECT_DIFF CURSOR FOR
 SELECT 'Object Differences' AS Object_Differences, A.*
 FROM QTEMP/OUTFILE1 A LEFT EXCEPTION JOIN
      QTEMP/OUTFILE2 B USING(ODOBNM,ODOBTP,ODOBAT)
 UNION ALL
 SELECT 'Object Differences' AS Object_Differences, B.*
 FROM QTEMP/OUTFILE1 A RIGHT EXCEPTION JOIN
      QTEMP/OUTFILE2 B USING(ODOBNM,ODOBTP,ODOBAT);
```

Note that in the second subquery, I am projecting OUTFILE2 columns using the B.* table alias reference.

Additional enhancements

I can picture a system administrator wanting this kind of compare, and wanting to stick to a green-screen instead of using the Run SQL Scripts tool. In that case, he may want to change the cursor declarations and create output tables instead, using the CREATE TABLE AS (…) WITH DATA syntax. This option would work with RUNSQLSTM script as well, not just the stored procedure (though the stored procedure has an added advantage of accepting library names as input arguments).

The object differences compare is overly simple in my example, especially in light of fact that I'm already comparing file objects using the SYSTABLES catalogue. While a simple option would be to remove the SYSTABLES compare, I think a better option is to be more selective about the DSPOBJD object type list. Instead of specifying *ALL as I have done, just specify non-table object types that matter in your environment. The reason I recommend this is that DSPOBJD against a library with many objects will be a long running operation, while the query against SYSTABLES will run relatively quickly.

Note that I have not included all possible SQL objects in the stored procedure, so you may want to complete the list and include them (i.e. SYSINDEXES, SYSSEQUENCES etc.).

Conclusion

This article illustrates the power of an EXCEPTION JOIN and how easy it is to leverage for administrative task such as comparing differences between two libraries, while touching on concepts such as FULL EXCEPTION JOIN, the EXCEPT clause and stored procedures returning multiple result sets.

I hope it spurs your creative juices into action.

Centerfield Technology

www.centerfieldtechnology.com

Centerfield Technology, Inc.
3131 Superior Drive NW
Suite C
Rochester, Minnesota USA 55901
888.387.8119 (toll free)
507.287.8119 (phone)