Decoupling RPG database IO using Rational Open Access: RPG Edition

A fresh start for RPG, ILE, and SQL

Daniel R. Cruikshank

September 06, 2011

Moving from a DDS to SQL database on DB2 for i can be accomplished without changing a single line of program code or recompiling a program. In this article I will describe how to use Rational Open Access: RPG Edition to take advantage of advanced data centric programming techniques only available via SQL programming.

IBM i customers, worldwide, are now using SQL to define and access data. With each new release, IBM continues to provide a wealth of new function and capability that can only be leveraged with SQL.

These new SQL based applications are taking advantage of modern data centric development concepts that stress the importance of pushing more of the business rules and logic into the database.

Many of these customers still rely upon heritage applications written in IBM's RPG programming language. Several of these RPG programs would also benefit from utilizing modern data centric techniques. These existing RPG applications contain the same business rules as the new SQL applications, however they were written many years ago using less efficient traditional record at a time data access methods. Duplication of rules and logic can (and does) result in expensive dual maintenance and multiple versions of the truth.

Enter Rational Open Access: RPG Edition or ROA. This support (introduced with IBM i 7.1 and retrofitted to support 6.1) can eliminate the high cost of dual maintenance and/or expensive program rewrites by allowing RPG IO operations to be intercepted and transformed into SQL. A new keyword (HANDLER) can now be added to an existing RPG file specification. The handler keyword specifies either a program or service program entry point that is called by the IBM i operating system. This new support can substantially reduce the amount of legacy code rewrite that would be required to convert existing programs to share common modules. An example of coding the handler is shown in Listing 1.

Listing 1: Example of coding the HANDLER keyword

```
FEMPADDRSL1UF E K DISK //-------------------Rational Open Access Addition ---------------------- // The
handler keyword specifies the program/service program which will // now process the IO operations for this
file F handler('UPDHANDLER')
```

In this article, I will describe how to create a format-based handler that allows multiple programs which are updating the same table using a common format and unique key, to take advantage of a single generic SQL UPDATE statement which utilizes extended indicator variable support.

The following is the approach that I will use:

- Build a format-based handler program that intercepts RPG traditional record at a time database IO operations. The handler program will convert the intercepted IO operation to equivalent SQL statements.
- Enable an existing non-SQL RPG ILE program to use the handler program by adding a single line of code.
- Create and register an external stored procedure which allows any interface (Java, PHP, etc.) that supports stored procedure calls to access the handler program

I have chosen RPG IV free format style as my host based programming language. This is due mainly to the fact that I believe the use of Rational Open Access: RPG Edition will appeal more to RPG programmers. However, it is important to note that the handler and SQL programs can be written in any host based language supported on IBM i.

I have also taken some liberties with the code examples, consolidating procedures, moving and removing code etc. This may result in the code examples not compiling as is. In addition, some code examples will only work if the Rational Open Access product is installed.

Building a Format-based Handler Program

There are two methods for passing data back and forth between the RPG program and the handler program using Rational Open Access. The first is structure-based whereas the data and key values are passed using buffers. These buffers can be program-described or externally-described. The second method is column-based where the data and key values are passed as individual items. Each item contains the attributes of the data or key field. The method used has a direct bearing on the number of handlers that may need to be written. In either case, the use of dynamic embedded SQL will be required to minimize the coding effort and overall number of handler programs.

Dynamic SQL is an ideal candidate for creating generic database handlers. There are three types of generic database handlers that you can begin to deploy depending on your level of expertise with ILE and SQL. The three types are format, statement or procedure-based handlers. The format-based handler lends itself well to fixed-list style dynamic SQL. The statement-based handler is better suited for varying-list style dynamic SQL. The procedure-based handler is best used for result set consumption or "one and done" type SQL statements such as MERGE.

Fixed-list dynamic SQL is typically used when an application provides the capability to dynamically change row selection (via the WHERE clause) or column ordering (via the ORDER BY clause) however the projected result set is static. In the handler application, fixed-list dynamic SQL would be used with the structure based method. An SQL descriptor is not required when using fixed-list dynamic SQL.

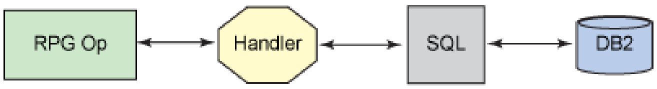
Varying-list dynamic SQL is typically used when an application builds the entire SQL statement from scratch. In this case, an SQL descriptor is required to describe the SQL statement, the columns projected in the result set along with any input parameters. There are two types of descriptors, one created via the SQL ALLOCATE DESCRIPTOR and the other is defined based on the SQLDA structure. Varying-list dynamic SQL would be used with the statement-based handlers. To learn more about dynamic SQL, reference the SQL Programming guide online at: http://ibm.com/systems/i/db2/books.html

For this article, I will be using a combination of fixed-list dynamic SQL and static embedded SQL (for the UPDATE statement) to create a format-based handler. In future articles, I will provide the makings for statement- and procedure-based handlers.

A major benefit of using the ROA product is that it eliminates the costly time and effort of modifying and testing an existing program. This approach to decoupling IO can provide a "fresh start" for developers tasked with maintaining existing RPG code which may not be well-structured or taking advantage of the modular capabilities provided by ILE (Integrated Language Environment) programming model.

Figure 1 provides an overview of how the handler program provides a bridge to an existing set of procedures that rely on SQL for data access. These procedures can be external stored procedures using embedded SQL or written entirely in SQL PL. In this article, I will be using RPG sub-procedures using embedded SQL.

Figure 1: Using a Handler to Decouple IO



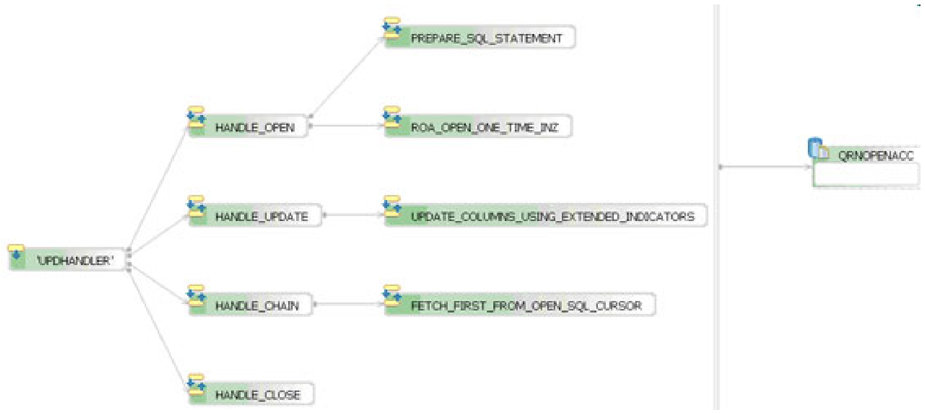
The handler uses ILE to provide a prototyped interface to the RPG program. In order to use a handler, the existing RPG program must be an ILE program also known as RPG IV. Thus, if an existing RPG program is not ILE then the first step would be to convert the RPG program to ILE. This is accomplished by the using the Convert RPG Source (CVTRPGSRC) command.

My handler program consists of a main module which directs the RPG operation to a corresponding handler module. This allows the main module to be customized by application type. For example, a handler for input-only files doesn't need the update modules.

The individual handler modules contain the logic for transforming the RPG IO operation to the appropriate SQL alternative. The transformation code is contained in sub-procedures. In addition, the handler modules contain the prototyped calls to an RPG SQL service program which contains the embedded SQL sub-procedures. Currently there are 19 database related RPG IO operations that may eventually require handling by your handler program. In my example handler program, I am using four of these, OPEN, CHAIN, UPDATE and CLOSE.

Figure 2 contains the Rational Visualize Application Diagram depicting the above scenario. The handler main procedure, UPDHANDLER, hands off the incoming RPG IO operation to the appropriate sub-procedure. The sub-procedure performs the transformation logic before calling the corresponding RPG SQL service program sub-procedure.

Figure 2: RDP Visualize Application Diagram



Larger view of Figure 2.

The Handler Main Procedure (UPDHANDLER)

The format-based handler will need to define the record format of the file being handled. This is done by coding templates for the file definition, file record format, file keys and SQL null indicator array as part of the handler module as shown in Listing 2. These are coded as RPG global variables to allow subsequent reuse of the generic template names and minimize code modification.

Listing 2: Defining the Record Format Template

```
FrcdFile_t IF E K DISK TEMPLATE F EXTDESC('EMPADDRESS') F RENAME(empAddr:rcdFormat) D rcdFormat_t... D DS
LIKEREC(rcdFormat) D TEMPLATE D keys_t DS LIKEREC(rcdFormat:*KEY) D TEMPLATE D Ind_Array_t... D S 5i 0 DIM(7)
D TEMPLATE
```

Listing 2 contains a code example of defining a file template. The TEMPLATE keyword, new in IBM i 6.1, informs the RPG compiler that this file will be used for field definition only, thus no IO operations are required or allowed for this file. The EXTDESC keyword is not required. In this case it is used to define the real file containing the record format definition used in the handler program.

The RENAME keyword is used to provide a generic format name to reduce code changes should this code be used as a template for other format handlers. This can be seen in the rcdFormat_t data structure template which is based on the renamed format. This data structure template is used in other modules within the handler program.

The SQL null indicator array is used in the RPG HANDLE_CHAIN and RPG HANDLE_UPDATE sub-procedures. The null indicator values are populated during the execution of the SQL FETCH statement. The null indicator values are set by the RPG HANDLE_UPDATE sub-procedure prior to executing the SQL UPDATE statement.

Once the handler keyword is specified for a file, all explicit or implicit IO operations executed against that file are now handled by the handler program. It is the programmer's responsibility to code the necessary instructions to process the IO operation and to provide the appropriate results. The ROA product is shipped with an RPG include file (QRNOPENACC) which contains the subfield definitions of the parameter that is passed to the handler program.

Listing 3 contains an example of the main procedure in the UPDHANDLER program. The input parameter (rpgIO) is a data structure defined like the QrnOpenAccess_t template which is part of the QRNOPENACC include module. The main procedure acts on the following subfields:

- The rpgStatus field is used for input and output to indicate success or failure. A zero indicates the operation was successful. The handler supplies a valid file status code if an exception occurs. The status codes are defined in the ILE RPG Reference Manual.
- The rpgOperation field contains a code corresponding to the RPG operation which was just performed by the RPG program being handled. The include module provides named constants for each IO operation code (i.e. QrnOperation_OPEN, QrnOperation_CHAIN, etc). The RPG Select statement controls which sub-procedure will be executed based on the RPG operation. The rpgIO data structure is passed to each sub-procedure and the rpgStatus field is used as a return field. If a sub-procedure sets rpgStatus to 1299 then an unrecoverable error has occurred and the handler is deactivated (*INLR is turned on).

Listing 3: Handler Main Procedure RPG Code Example

```
D UPDHANDLER PI D rpgIO... D LIKEDS(QrnOpenAccess_T) /COPY QOAR/QRPGLESRC,QRNOPENACC /FREE rpgIO.rpgStatus
= *Zero; select; when rpgIO.rpgOperation = QrnOperation_OPEN; rpgIO.rpgStatus = Handle_Open(rpgIO); when
rpgIO.rpgOperation = QrnOperation_CHAIN; rpgIO.rpgStatus = Handle_Chain(rpgIO); when rpgIO.rpgOperation =
QrnOperation_UPDATE; rpgIO.rpgStatus = Handle_Update(rpgIO); when rpgIO.rpgOperation = QrnOperation_CLOSE;
rpgIO.rpgStatus = Handle_Close (rpgIO); Other; ENDSL; //If unrecoverable error then shutdown handler If
rpgIO.rpgStatus = 1299; *INLR = *On; ENDIF; return; /END-FREE
```

The Sub-procedure Interface

Each RPG IO operation sub-procedure uses a common interface consisting of the rpgIO parameter and a return field corresponding to the rpgStatus code. In addition, each procedure sets rpgStatus to zero upon entry and uses the RPG Monitor feature to handle unexpected errors. When such an error occurs, the rpgStatus field is set to 1299 (Other I/O error detected). This causes an exception to occur in the RPG program being handled.

Listing 4 contains a code snippet of the common handler sub-procedure interface.

Listing 4: Common Sub-procedure Interface Prototype

```
P Handle_Open B D Handle_Open PI LIKE(rpgIO.rpgStatus) D rpgIO... D LIKEDS(QrnOpenAccess_T) D* Local fields
D retField S LIKE(rpgIO.rpgStatus) /FREE retField = *Zero; Monitor; //routine specific data and code begins
here On-Error; retField = 1299; ENDMON; RETURN retField; /END-FREE P Handle_Open E
```

Handling Implicit and Explicit RPG Open Operations

The RPG file open can occur implicitly during the RPG initialization phase or explicitly via the RPG OPEN operation and the use of the USROPN keyword on the file definition. In most cases, implicit opens will be sufficient. The use of the explicit OPEN operation provides more interaction between the RPG program and the handler.

The first thing that should be done in the handler OPEN routine is to set the pointers and lengths of the IO Information Feedback data structures. These are the structures that can be used in an RPG program when unexpected errors occur. It allows you to send additional status information back to the RPG program. Listing 5 contains a code snippet for setting these values in the rpgIO data structure.

Listing 5: IO Feedback Information

```
rpgIO.openFeedback = %Alloc(80); rpgIO.ioFeedback = %Alloc(160); rpgIO.deviceFeedback = %Alloc(126);
rpgIO.openFeedbackLen = 80; rpgIO.ioFeedbackLen = 160; rpgIO.deviceFeedbackLen = 126;
```

The next item of importance is the rpgIO stateInfo pointer. This optional pointer is used to allocate temporary storage where you can store information that needs to be retained between calls to and from the handler. One such item is a before record image. The handler program is going to compare the before record image with the values passed into the handler from the RPG update operation. We will cover more on this in the Handle-Update sub-procedure discussion.

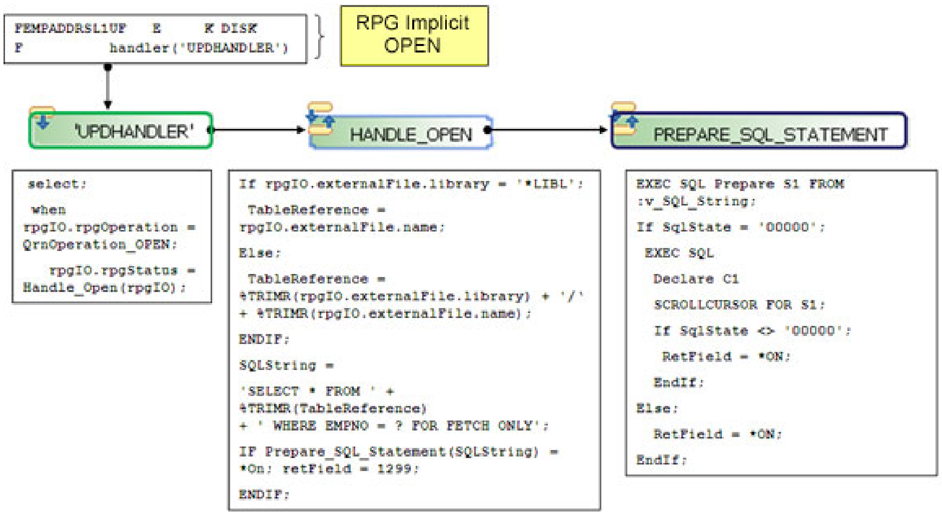
Listing 6 contains a code example of declaring a state information structure and then allocating storage for the stateInfo pointer based on the size of the structure.

Listing 6: Code example for the stateInfo data structure

```
//stateinfo data structure template D rpgSI_t... D DS TEMPLATE D OLD_ROW_p * /FREE rpgIO.stateInfo =
%Alloc(%Size(rpgSI_t));
```

At this point, we are ready to construct the SQL statement that will be used to return the input record in response to the RPG CHAIN operation. This statement will be a string variable that is passed to the Prepare_SQL_Statement sub-procedure. Figure 3 details the process of handling an RPG implicit file open operation along with code snippets of the major functions. The name of the file opened by the RPG program is passed to the handler in the rpgIO externalFile structure. This structure consists of two subfields: library and name. If the library column contains *LIBL, then only the external file name is used for the table-reference otherwise the library and name columns are combined to form a qualified table-reference. The table-reference variable is then used for the SQL Select statement FROM clause. When the call to the RPG PREPARE_SQL_STATEMENT sub-procedure fails, an rpgStatus value of 1299 will be returned to the RPG program being handled. This is shown in the code snippet located under the HANDLE_OPEN process icon in Figure 3

Figure 3: Handling RPG Open Operations



```
FEMPADDRSL1UF   E       K DISK
F            handler('UPDHANDLER')
```

```
select;
 when
rpgIO.rpgOperation =
QrnOperation_OPEN;
   rpgIO.rpgStatus =
Handle_Open(rpgIO);
```

```
If rpgIO.externalFile.library = '*LIBL';
 TableReference =
rpgIO.externalFile.name;
Else;
 TableReference =
%TRIMR(rpgIO.externalFile.library) + '/'
+ %TRIMR(rpgIO.externalFile.name);
ENDIF;
SQLString =
'SELECT * FROM ' +
%TRIMR(TableReference)
+ ' WHERE EMPNO = ? FOR FETCH ONLY';
IF Prepare_SQL_Statement(SQLString) =
*On; retField = 1299;
ENDIF;
```

```
EXEC SQL Prepare S1 FROM
:v_SQL_String;
If SqlState = '00000';
 EXEC SQL
  Declare C1
  SCROLLCURSOR FOR S1;
  If SqlState <> '00000';
   RetField = *ON;
  EndIf;
Else;
  RetField = *ON;
EndIf;
```

A WHERE clause is added to the Select statement string representing the column or columns that correspond to the key fields used in the RPG CHAIN operation. In this example, EMPNO is the unique key of the table. At this time the value of the EMPNO is not known so a parameter marker (?) is used as a placeholder. The actual value will be provided on the first RPG keyed IO operation. A FOR FETCH ONLY clause is added to the Select statement to avoid record locking and to take advantage of SQL automatic blocking.

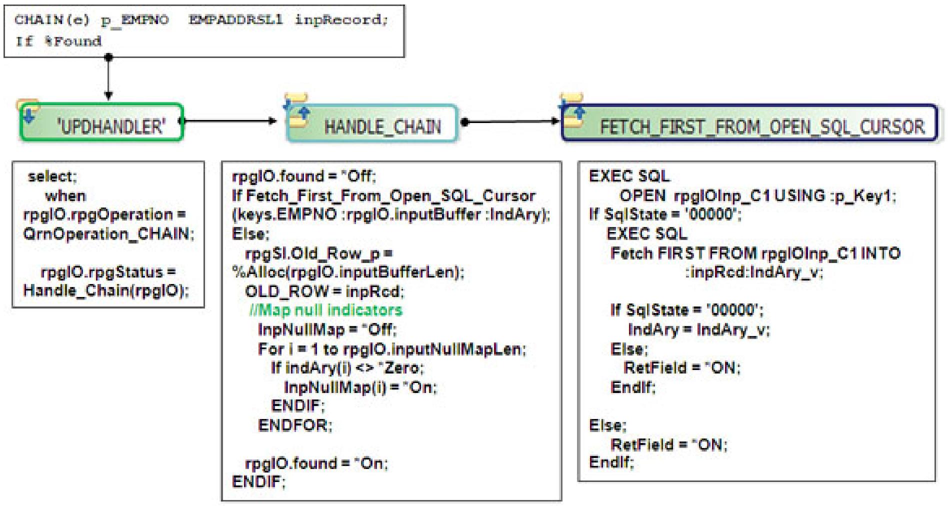
Once the SQL statement formatting is complete it is passed to the RPG PREPARE_SQL_STATEMENT sub-procedure. The RPG procedures containing the embedded SQL statements are normally created as separate modules and then used to create a service program. This service program can be bound to the handler program at compile time. The SQL module must also contain the format template code as shown in Listing 2. In addition, to support the usage of SQL extended indicators, the SQL module either needs to be compiled with the *EXTIND on the OPTION parameter of the precompiler command, or contain an SQL Set Option statement specifying EXTIND(*YES). The Set Option statement needs to be specified before any other SQL statement within the module.

Figure 3 contains an RPG code example for the RPG PREPARE_SQL_STATEMENT sub-procedure which prepares a dynamic SQL statement string (v_SQL_String) and, if successful, declares an SQL cursor for the prepared SQL statement. The SQL statement string (p_SQL_String) is passed to the RPG sub-procedure from the RPG HANDLE_OPEN sub-procedure in response to the RPG Open operation.

Handling the RPG CHAIN Operation

Before an RPG program updates a row accessed by a key it must perform a read operation to lock the row for update intent. The most common RPG method for performing a random read using a key is the CHAIN operation. Figure 4 details the process of handling an RPG CHAIN operation along with code snippets of the major functions.

Figure 4: Handling RPG CHAIN Operations



```
CHAIN(e) p_EMPNO  EMPADDRSL1 inpRecord;
If %Found
```

```
 select;
   when
rpgIO.rpgOperation =
QrnOperation_CHAIN;

   rpgIO.rpgStatus =
Handle_Chain(rpgIO);
```

```
rpgIO.found = *Off;
If Fetch_First_From_Open_SQL_Cursor
(keys.EMPNO :rpgIO.inputBuffer :IndAry);
Else;
  rpgSI.Old_Row_p =
%Alloc(rpgIO.inputBufferLen);
  OLD_ROW = inpRcd;
   //Map null indicators
    InpNullMap = *Off;
    For i = 1 to rpgIO.inputNullMapLen;
      If indAry(i) <> *Zero;
        InpNullMap(i) = *On;
      ENDIF;
    ENDFOR;

  rpgIO.found = *On;
ENDIF;
```

```
EXEC SQL
     OPEN  rpgIOInp_C1 USING :p_Key1;
If SqlState = '00000';
   EXEC SQL
    Fetch FIRST FROM rpgIOInp_C1 INTO
                 :inpRcd:IndAry_v;

    If SqlState = '00000';
      IndAry = IndAry_v;
    Else;
      RetField = *ON;
    EndIf;

Else;
    RetField = *ON;
EndIf;
```

When using the structure based IO method, RPG will create the initial storage for the input and output buffers, along with storage for the null indicator map. The rpgIO parameter contains pointers to these storage areas. The RPG HANDLE_CHAIN sub-procedure defines data structures like the global templates defined earlier. The data contained within the structures are based on the pointers provided by RPG. Listing 7 provides the source code examples for the data structure definitions. Figure 4 contains a sample of the RPG source code for the RPG HANDLE_CHAIN sub-procedure.

Listing 7: Handling the RPG CHAIN

```
D Handle_Chain PI LIKE(rpgIO.rpgStatus) D rpgIO LIKEDS(QrnOpenAccess_T) D i S 5i 0 D keys DS
LIKEDS(keys_t) D BASED(rpgIO.key) D inpRcd DS LIKEDS(rcdFormat_t) D BASED(rpgIO.inputBuffer) D OLD_ROW DS
LIKEDS(rcdFormat_t) D BASED(rpgSI.OLD_ROW_p) D rpgSI... D DS LIKEDS(rpgSI_t) D BASED(rpgIO.stateInfo) D
IndAry S 5i 0 DIM(%elem(Ind_Array_t)) D InpNullMap S N DIM(%elem(Ind_Array_t)) D BASED(rpgIO.inputNullMap)
```

The inputBuffer pointer will be set to nulls by RPG once the data is received by the RPG program. In order to compare the before and after record images the handler must make a copy of the input record. The storage for the old row is defined as part of the stateInfo data structure described in Listing 6. This ensures the data is preserved between the CHAIN and subsequent UPDATE operations.

If the column is defined as null capable, then you must update the null indicator map accordingly. RPG uses a 1 character field (like an indicator) to determine if a column contains nulls. SQL uses a small integer which contains a zero if the column is not null or -1 if the column contains nulls.

In SQL, there is no equivalent for the RPG CHAIN so I have chosen to use the FETCH FIRST statement with a cursor as the appropriate alternative. This also explains the use of the PREPARE, DECLARE, OPEN and CLOSE SQL cursor statements.

One school of thought might suggest the use of the SELECT INTO as an alternative to the CHAIN operation. In some ways it is similar to an RPG program using CHAIN as it implicitly performs the preparation, open and close functions; however the disadvantages far outweigh this shorthand capability. These disadvantages are:

1. The SELECT INTO must return a single row or it fails. There is no guarantee that the RPG program is using a unique key on the CHAIN. In this case the row returned to RPG is dependent on which of the duplicate key handling DDS keywords (LIFO (default), FIFO or LCFO) were used when creating the keyed logical file.
2. The above problem can be circumvented by the use of DISTINCT and one or more timestamp columns to simulate FIFO or LCFO; however DB2 will use an additional step to eliminate the duplicates. This could result in a significant decrease in performance should there be a large number of duplicate key values.
3. Because the SELECT INTO returns a single row it cannot be used for other RPG read operations. Whereas the FETCH can be used to return 1 or more rows allowing a single RPG sub-procedure to be used for CHAIN, READ or READE operations.

Figure 4 contains a sample of the RPG source code for the FETCH_FIRST_FROM_OPEN_CURSOR sub-procedure. The RPG HANDLE_CHAIN sub-procedure passes the key (or keys), the inputBuffer pointer and the indicator array.

The key (or keys) are used to replace the parameter marker (or markers) which were defined as part of the SQL statement string created in the RPG HANDLE_OPEN sub-procedure. The OPEN CURSOR statement is executed as part of the FETCH process. If the OPEN is successful then the SQL FETCH FIRST statement is executed. The inpRcd structure and indAry parameters contain the results of a successful FETCH. The combination of the SQL OPEN and FETCH is equivalent to system instructions performed on behalf of the RPG CHAIN. In essence, the OPEN positions into the index and the FETCH FIRST will retrieve the first row based on the RRN provided by the index.

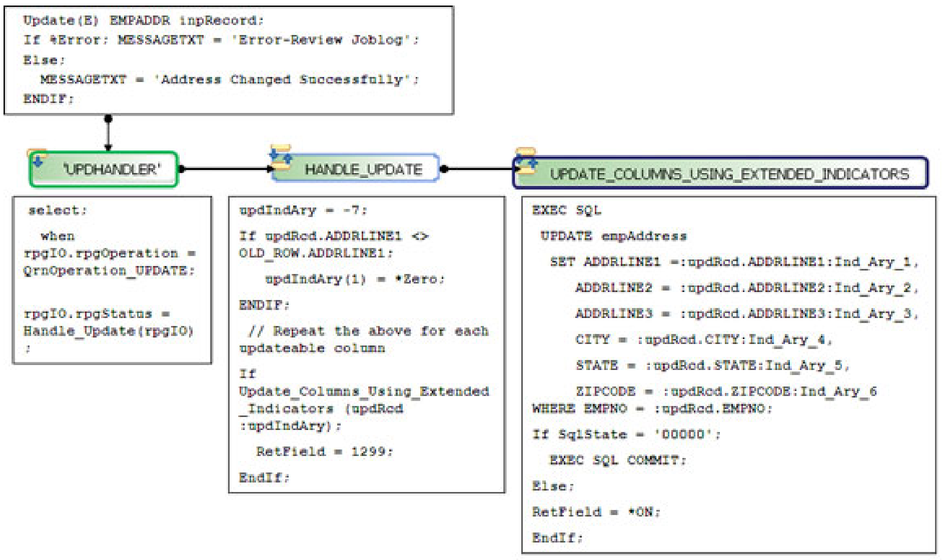
After the second execution of the OPEN within the same session or job, the SQL cursor will become reusable. This means that subsequent SQL OPEN and CLOSE operations will merely reposition the cursor to the first row of the result set. This behavior will occur regardless of whether or not the RPG program is using the LR indicator. This allows the continued use of LR as a housekeeping tool while avoiding the high cost of SQL open and close overhead.

Handling the RPG UPDATE Operation

To understand the RPG HANDLE_UPDATE sub-procedure we need to discuss the concept of extended indicator variable support. Prior to the DB2 for i 6.1 release, SQL Update and Insert statements could use an indicator variable with a value of -1 to set a null capable column to the null value. With extended indicator variable support, introduced in 6.1, you can extend update and insert capabilities by providing additional indicator values.

Of most interest is the ability to use an indicator value of -7 to allow an update to bypass the column as if it was not a part of the UPDATE statement. This support allows you to write a generic update procedure that can be used for any update transaction, regardless of the columns being updated. Figure 5 details the process of handling an RPG UPDATE operation along with code snippets of the major functions.

Figure 5: Handling RPG Update Operation



```
Update(E) EMPADDR inpRecord;
If %Error; MESSAGETXT = 'Error-Review Joblog';
Else;
  MESSAGETXT = 'Address Changed Successfully';
ENDIF;
```

```
 select;
   when
rpgIO.rpgOperation =
QrnOperation_UPDATE;

rpgIO.rpgStatus =
Handle_Update(rpgIO)
;
```

```
updIndAry = -7;
If updRcd.ADDRLINE1 <>
OLD_ROW.ADDRLINE1;
   updIndAry(1) = *Zero;
ENDIF;
 // Repeat the above for each
updateable column
If
Update_Columns_Using_Extended
_Indicators (updRcd
:updIndAry);
  RetField = 1299;
EndIf;
```

```
EXEC SQL
 UPDATE empAddress
  SET ADDRLINE1 =:updRcd.ADDRLINE1:Ind_Ary_1,
     ADDRLINE2 = :updRcd.ADDRLINE2:Ind_Ary_2,
     ADDRLINE3 = :updRcd.ADDRLINE3:Ind_Ary_3,
     CITY = :updRcd.CITY:Ind_Ary_4,
     STATE = :updRcd.STATE:Ind_Ary_5,
     ZIPCODE = :updRcd.ZIPCODE:Ind_Ary_6
WHERE EMPNO = :updRcd.EMPNO;
If SqlState = '00000';
  EXEC SQL COMMIT;
Else;
RetField = *ON;
EndIf;
```

The update indicator array is initialized to -7 as shown in the Figure 5 code snippet from the RPG HANDLE_UPDATE sub-procedure. The values in the outputBuffer are compared to the values that were saved from the inputBuffer. The indicator variable for the column is set to zero for each value that is different. Additional code could be added to perform a check to see if the new value is a user-defined value indicating that the database value should be set to NULL or the default value defined for the column. If the former is true, then the indicator variable for that column is set to -1 which results in the column being updated with the null value. If the latter is true, then the indicator variable for that column is set to -5 which will set the column to the default value for that column. If the old and new values are the same, then the indicator variable remains as -7 and the column is ignored. The RPG HANDLE_UPDATE sub-procedure is format specific and must be customized for each file.

When all columns have been compared the RPG UPDATE_COLUMNS_USING_EXTENDED_INDICATORS sub-procedure is executed. The RPG UPDATE_COLUMNS_USING_EXTENDED_INDICATORS sub-procedure accepts two parameters, updRcd and Ind_Ary. The updRcd parameter contains the values (both changed and unchanged) for the updateable columns in the table. The Ind_Ary column contains the SQL extended indicator settings.

Figure 5 contains a sample of the RPG source code for the RPG UPDATE_COLUMNS_USING_EXTENDED_INDICATORS sub-procedure. The table name used on the UPDATE is the name of the file containing the format used in Listing 2. In the IBM Data Access Reengineering strategy, this file is referred to as the surrogate file.

The indicator variables can not be specified using the array indexing technique. Each indicator variable must be individually named. To get around this I use a data structure to define a named indicator variable for each array element. The data structure is based on the address of Ind_Ary. Listing 8 contains the RPG source code example for redefining the input indicator array to a data structure.

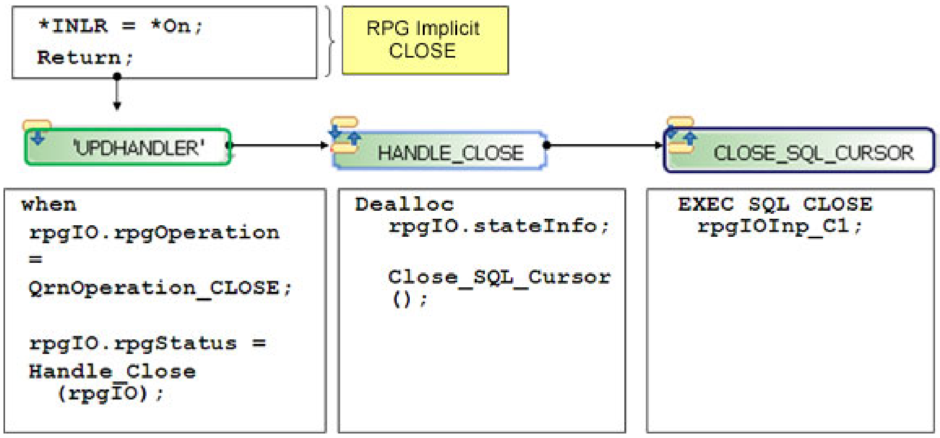
Listing 8: Update Procedure with Extended Indicators

```
D Update_Columns_Using_Extended_Indicators... D PI N D updRcd... D LIKEDS(rcdFormat_t) D Ind_Ary... D
LIKE(Ind_Array_t) D DIM(%elem(Ind_Array_t)) D* Local fields D retField S N D Ind_Ary_DS DS BASED(Ind_Ary_Ptr)
D Ind_Ary_1 5i 0 D Ind_Ary_2 5i 0 D Ind_Ary_3 5i 0 D Ind_Ary_4 5i 0 D Ind_Ary_5 5i 0 D Ind_Ary_6 5i 0 D
Ind_Ary_7 5i 0 /FREE Ind_Ary_Ptr = %Addr(Ind_Ary);
```

Handling Implicit and Explicit RPG Close Operations

The handler program must perform two functions: 1) close the SQL cursor and 2) deallocate the memory used by the stateInfo data structure. Figure 6 details the process of handling an implicit or explicit RPG CLOSE operation along with code snippets of the major functions.

Figure 6: Handling RPG CLOSE Operations



```
*INLR = *On;
Return;
```

```
when
 rpgIO.rpgOperation
 =
 QrnOperation_CLOSE;

 rpgIO.rpgStatus =
 Handle_Close
   (rpgIO);
```

```
Dealloc
   rpgIO.stateInfo;

   Close_SQL_Cursor
   ();
```

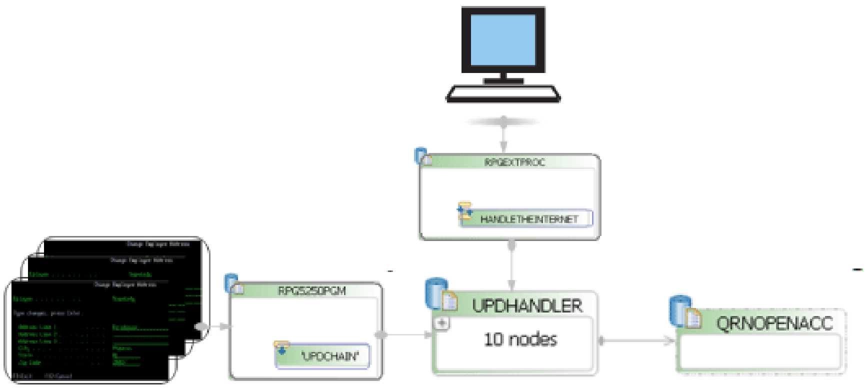
```
EXEC SQL CLOSE
  rpgIOInp_C1;
```

The RPG program turns on the Last Record (*INLR) indicator which results in an implicit CLOSE operation against the handled file. The handler intercepts the RPG CLOSE operation and passes control to the RPG HANDLE_CLOSE sub-procedure. The memory allocated to the stateInfo pointer is deallocted. A call is issued to the RPG CLOSE_SQL_CURSOR sub-procedure which contains the embedded SQL CLOSE statement.

Implementation Scenarios

The following scenarios were used to test the handler code: 1) a traditional RPG program using a 5250 Display File and 2) an external stored procedure called from a Java application. The 5250 scenario is typical of many traditional RPG shops where many programs are accessing the same table for update. The Java scenario describes a technique where a single external stored procedure is used for table updates versus multiple SQL UPDATE statements contained in many applications. The goal is to have a single point of control for all updates to the same table regardless of the point of origin. These scenarios are depicted in Figure 7.

Figure 7: Mixed Apps Sharing Common Handlers



In essence, a single format-based handler program is able to service multiple host based RPG 5250 programs using a variety of UPDATE methods. This includes updating the file or record format using data structures or the RPG built-in function %FIELDS. In addition, once an RPG program is registered as an external stored procedure, any interface (Java, PHP, etc.) that supports stored procedure calls can access the handler program. This external stored procedure approach also allows external applications such as a browser client to also take advantage of DB2 for i extended indicator support.

Listing 9 contains a code example of the SQL CREATE PROCEDURE statement used to register an RPG program as an external stored procedure.

Listing 9: Code Example of External Stored Procedure

```
CREATE PROCEDURE TEST_UPDFIELDS ( IN p_ADDRESS1 VARCHAR(30) ,IN p_ADDRESS2 VARCHAR(30) ,IN p_ADDRESS3
VARCHAR(30) ,IN p_CITY VARCHAR(15) ,IN p_STATE VARCHAR(10) ,IN p_ZIPCODE VARCHAR(10) ,IN Unique_Key CHAR(6) )
LANGUAGE RPGLE PARAMETER STYLE GENERAL WITH NULLS RESULT SETS 2 NOT DETERMINISTIC MODIFIES SQL DATA EXTERNAL
NAME RPGEXTPROC
```

Listing 10 contains the code for the existing RPG program that has been registered as an external stored procedure. The RPG IO operation codes that will be handled by the UPDHANDLER program are **BOLD**. The RPG program is defining the surrogate file for update intent. The PARAMETER STYLE GENERAL WITH NULLS clause from the SQL CREATE PROCEDURE statement shown in Listing 9 specifies that an array of null indicators will be sent as an additional parameter to the external stored procedure. There will be one element in the array for each input parameter passed to the external stored procedure. The RPG program uses this parameter (p_Ind_Ary) to determine if the input parameter contains a value other than NULL (-1). If so, then the corresponding data column is changed with the input parameter value. Since the file EMPADDRESS is being handled by program UPDHANDLER, then the UPDATE operation is processed as described earlier in Figure 5.

Listing 10: Code Example of RPG External Procedure

```
 FEMPADDRESSUF E K DISK F handler('UPDHANDLER') D i S 5i 0 D HandleTheInternet... D PR EXTPGM('RPGEXTPROC')
 D p_ADDRLINE1 LIKE(ADDRLINE1) D p_ADDRLINE2 LIKE(ADDRLINE2) D p_ADDRLINE3 LIKE(ADDRLINE3) D p_CITY
 LIKE(CITY) D p_STATE LIKE(STATE) D p_ZIPCODE LIKE(ZIPCODE) D p_EMPNO LIKE(EMPNO) D p_Ind_Ary... D 5i 0
 DIM(7) P HandleTheInternet... P B D HandleTheInternet... D PI D p_ADDRLINE1 LIKE(ADDRLINE1) D p_ADDRLINE2
 LIKE(ADDRLINE2) D p_ADDRLINE3 LIKE(ADDRLINE3) D p_CITY LIKE(CITY) D p_STATE LIKE(STATE) D p_ZIPCODE
 LIKE(ZIPCODE) D p_EMPNO LIKE(EMPNO) D p_Ind_Ary... D 5i 0 DIM(7) D inpRecord Ds LIKEREC(EMPADDR:*INPUT) /
Free Monitor; CHAIN(e) p_EMPNO EMPADDRESS inpRecord; If %Found; If p_Ind_Ary(1) = *Zero; inpRecord.ADDRLINE1
 = p_ADDRLINE1; ENDIF; If p_Ind_Ary(2) = *Zero; inpRecord.ADDRLINE2 = p_ADDRLINE2; ENDIF; If p_Ind_Ary(3) =
 *Zero; inpRecord.ADDRLINE3 = p_ADDRLINE3; ENDIF; If p_Ind_Ary(4) = *Zero; inpRecord.CITY = p_CITY; ENDIF;
 If p_Ind_Ary(5) = *Zero; inpRecord.STATE = p_STATE; ENDIF; If p_Ind_Ary(6) = *Zero; inpRecord.ZIPCODE =
 p_ZIPCODE; ENDIF; inpRecord.EMPNO = p_EMPNO; Update(E) EMPADDR inpRecord; ENDIF; On-Error; ENDMON; *INLR =
 *On; //Implicit CLOSE Return; /End-Free P HandleTheInternet... P E
```

Summary

In this article I introduced you to the concept of using Rational Open Access: RPG Edition to transform traditional record at a time access methods to take advantage of more advanced SQL techniques such as extended indicator variable support with as little as one line of code change in the existing RPG program.

With extended indicator support, a single program can be used for all incoming update transactions against a single table based on a common key. This includes existing programs using traditional update methods and external interfaces capable of utilizing stored procedure calls. This technique can eliminate the need to construct multiple SQL statements which update a subset of the columns within a table.

I also introduced the concept of a format-based handler which allows you to use the RPG input and output buffers as the mechanism for moving data between the RPG program and the handler. The format-based handler exploits the new template capability of RPG IV. It also builds upon the concept of externally described data structures, a way of soft coding the host variables used by fixed-list dynamic SQL.

In future articles, I will expand on these concepts by creating a statement-based handler that works in conjunction with varying-list dynamic SQL which can substantially reduce the number of handlers required. I will use these techniques to take advantage of the following advanced SQL capabilities:

- Bulk data handling techniques which utilize SQL blocked FETCH and INSERT to overcome traditional operations (i.e. READE) that cannot be blocked.
- Mass updates using SQL searched UPDATE and DELETE techniques
- Replacing RPG traditional archiving and purging techniques with an SQL MERGE statement
- Stored Procedure result set consumption
- Updateable SQL Join views using INSTEAD OF TRIGGER support

All of the above can be utilized with minimal change to existing programs via the IBM Rational Open Access: RPG Edition product.

Resources

You can find more information about this product at the following websites:

- http://www.ibm.com/software/rational/products/openaccess/
- http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp and then expand IBM i 7.1 Information Center->Programming->Programming Languages->RPG

To learn more about the Rational Developer for Power Systems tool used to create the Visualizer Application Diagrams within this article, and to download an evaluation copy, visit the developerWorks Rational Tools download website

Suggested Reading

- Case Study: Modernizing a DB2 for i Application

For a better understanding of the surrogate file concept mentioned in this article, and the IBM Database Modernization strategy, review the following Redbook:

- Modernizing iSeries Application Data Access (SG24-6393)

Visit the DB2 for i developerWorks forum to stay connected, ask questions and get solutions.