Out in Left Field
strategic tools for strategic teams
Volume Two, Issue Three

Performance Priorities

Over the years I have developed a set of rules to use when performance tuning is required. These rules can be used by anyone involved in the tuning process to help decide what approach is best in any given situation. They are generic, but straight to the point:

1. ***If a piece of work does not have to be done, don't do it.***
2. ***If the work must be done, only do it once.***
3. ***If the work must be done many times, do it efficiently.***

Most programmers start their performance tuning effort with number three. In other words, they take a look at a section of program code or an algorithm and try to figure out how to do it better. This is a natural tendency when working on a complex, mission critical application. Making local changes is always safer and generates fewer side-effects. Because of the reduced risk and localized impact, programmers will often polish a small piece of code for hours trying to get every possible CPU cycle eliminated. So why does this approach often lead to very small performance improvements?

The first reason is that often programmers are guessing at the reason for slow performance, relying on their intuition, which may or may not be accurate.

Secondly, CPU cycles are becoming an increasingly smaller part of an application's response time. In fact, delays caused by ***waits*** often impact an end-user's response time the most. Waits can occur at the communications level, because of locking, seizes, and disk I/O, to name a few.

The third reason for poor results is that even if a small part of the application takes 5% of the time, the best case scenario is that there will be 100% improvement in that five percent. Clearly most users will not notice a five percent improvement in their response time.

So let's take a closer look at the first and second prioritization rules.

The first rule tries to totally eliminate work or steps in a process. Let's say for example, that a report is generated every day and takes an hour to create. This report is part of a larger batch job that takes four hours to complete. The IT staff probably assumes that if the report is being generated, it is being used. With some simple questions, it might turn out that the report was needed three years ago but nobody ever looks at it any more because they can see the same information on-line. In this case, taking out the report generation would cut the batch job's elapsed time by 25%.

A second example of "work elimination" is the processing of relatively static data files. It is not uncommon to see batch jobs that process the same file routinely and generate summary tables or reports. If the file has not changed since running the previous batch stream, then it makes little sense to process the data again. Simply keeping the file's 'last changed' date and not processing the data if it has not changed is a very simple technique to avoid the unnecessary processing.

The best part about applying rule #1 is that 100% of the work is now saved. You simply can't achieve that kind of percentage using rule number three.

My second rule basically says that if work has been done and can be reused, then do it. A couple of easy examples can be seen in the world of SQL.. You may have a particular query that requires an index. This index gets built every time the query is run and then destroyed at the end. If the index were created as a permanent logical file or SQL index, the repetitive index build could be eliminated.

Another example of this technique is through the use of Materialized Query Tables (MQTs). MQTs allow the query optimizer to take advantage of 'pre-run' queries. The results for these queries are stored away in tables that look normal except the optimizer knows they are MQTs and therefore contain pre-calculated results that can be used. Essentially, part of the work that would have been done by the optimizer has already been completed and can be reused.

While you can't achieve a 100% gain as with rule number one, you can often see significant improvement if the work was removed from a loop in the program. The more times the program typically loops, the higher the percentage of improvement you will see because more aggregate work has been eliminated from the job.

So as you do performance tuning on your system or application, keep these rules in mind and you'll be able to deliver far better results than you have before.

Inside this issue:

Back to the FUTURE

A few years ago one of the hottest technologies around was something called OLAP, which stands for On-Line Analytical Processing. Essentially the benefit of OLAP databases was that a user could ask questions of the database and get nearly instantaneous response. The ability to query a large database and get real-time answers made OLAP the darling of many data warehouse experts.

So how did this technology really work? The answer to this question varied somewhat based on which vendor's solution you chose to look at, but the most popular products pre-calculated results and put those answers into a proprietary data structure. The data storage was then accessed through APIs driven from a graphical interface or spreadsheet macros. OLAP databases provided several advantages over relational queries. By design, the databases were created for business analysis and so were more intuitive to work with for non-IT professionals. Typically thousands of queries were pre-calculated so the speed allowed business analysts to do their work without the delay associated with typical reporting environments.

The disadvantages of OLAP engines centered on the need to extract the data from relational databases, generate the data for thousands of queries, and store that data. This process was often complex, time consuming and expensive in terms of disk storage.

Recognizing the need to speed up query processing, several database vendors applied the OLAP concept to the relational model. Known as summary tables or materialized views, pre-calculated queries could be stored in the database and automatically used by the query optimizer to speed up SQL statements.

As of V5R3 (with PTF SI17164 and the latest database group PTF), IBM introduced the concept into DB2 for the iSeries and i5 and dubbed the support Materialized Query Tables. Here are the basic steps to create and use MQTs:

- Determine what queries or parts of queries are calculated often.
- Create an SQL table tagged as an MQT that satisfies the query.
- Provide a job permission to use MQTs via the query options table (QAQQINI)
- Submit a query with syntax that can take advantage of pre-calculated results and with options that allow copies of data to be used
- Refresh the MQT data on a regular basis so that incoming queries do not use data that is too stale.

Depending on the query or the frequency of common SQL syntax, MQTs can provide very significant performance benefits. In a later column, we'll get into more details about when MQTs can be used and what it takes to get the most bang for the buck.

Using DBMON to tune non-SQL queries

Database Monitor is primarily designed for tuning SQL queries. This is evident in iSeries Navigator Performance Monitor queries, associated literature and DBMON data itself.

It is a little known fact though, that it is possible to tune non-SQL queries using database monitor as well. A number of non-SQL interfaces access the System i database via the query optimizer. The most well known non-SQL interfaces are OPNQRYF, QUERY/400 (WRKQRY,RUNQRY) and QQQQRY API.

The base requirement for tuning non-SQL queries is to specify *DETAIL in the TYPE parameter of the STRDBMON command. This parameter indicates that detail rows as well as summary rows must be collected for fetch operations, with the Detail row designated by QQRID type of 3019. This is the only way to determine the number of rows that are returned and the total time to return those rows for non-SQL interfaces. Collecting this detailed information may cause slight performance degradation, but I am convinced it's well worth it, even when tuning purely SQL queries.

A large number of physical I/O operations can indicate that a larger pool is necessary or that SETOBJACC might be used to bring some of the data into main memory beforehand. To determine if that is an option to consider, I recommend the following query:

```
WITH qqq1000 AS (
  SELECT qqjfld,qqucnt
  FROM   dbmon
  WHERE  qqrid=1000 AND qqucnt<>0 AND qqc21<>'MT'),
retrieved AS (
  SELECT a.qqjfld,a.qqucnt,qqi2,qqi7
  FROM   dbmon a EXCEPTION JOIN qqq1000 b ON
         (a.qqjfld = b.qqjfld and a.qqucnt = b.qqucnt)
  WHERE  qqrid=3019)
SELECT  (qqi1+b.qqi2) "Total Query Time",
       b.qqi7 "Number Rows Retrieved",
       qqc101 "Open ID", qquser "Job User",
       qqjob "Job Name", qqjnum "Job Number"
FROM dbmon a INNER JOIN retrieved b ON
      (a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt)
WHERE qqrid=3014 AND SUBSTR(qqc101,1,1) NOT IN ('
','*',x'00')
ORDER BY "Total Query Time" DESC
```

You can use "Total Query Time" and "Number Rows Retrieved" ratios to calculate your row retrieval costs and then decide if using SETOBJACC to bring the data to the memory pool beforehand will benefit the query in question.
Notice that a lot of labor this query performs is to weed out SQL generated queries. To accomplish that I use an exception join on QQRID = 1000 since I know that non-SQL interfaces don't generate 1000 query id records. Based on the start and end time of a Database Monitor collection, it is still possible for some SQL-related queries to be included in our result set so we filter it further by checking the open ID's starting character for validity. Even with this additional check, it is possible for some SQL-related rows to appear in our result set. You might need to further customize the WHERE clause in the final select statement.

I realize that SETOBJACC is not a cure to all ailments pertaining to poorly performing non-SQL queries. So, let's try and address a scenario where a non-SQL query is causing table scans to be performed since appropriate indexes don't exist.

Table scan information is contained in rows with QQRID = 3000, so I am going to join that row with the result set generated by the base query listed above:

```
WITH qqq1000 AS (
  SELECT qqjfld,qqucnt
  FROM   dbmon
  WHERE  qqrid=1000 AND qqucnt<>0 AND qqc21<>'MT'),
retrieved AS (
  SELECT a.qqjfld,a.qqucnt,qqi2,qqi7
  FROM   dbmon a EXCEPTION JOIN qqq1000 b ON
         (a.qqjfld = b.qqjfld and a.qqucnt = b.qqucnt)
  WHERE  qqrid=3019),
index_adv AS (
  SELECT b.*, QQIDXA,qqidxd
  FROM   dbmon a INNER JOIN retrieved b on
         (a.qqjfld = b.qqjfld and a.qqucnt = b.qqucnt)
  WHERE  a.qqrid = 3000)
SELECT (qqi1+b.qqi2) "Total Query Time",
       b.qqi7 "Number Rows Retrieved",
       qqc101 "Open ID", qquser "Job User",
       qqjob "Job Name", qqjnum "Job Number",
       b.qqidxa "Index Advised",
       b.qqidxd "Columns Advised"
FROM dbmon a INNER JOIN index_adv b ON
       (a.qqjfld = b.qqjfld AND a.qqucnt = b.qqucnt)
WHERE qqrid=3014 AND SUBSTR(qqc101,1,1) NOT IN ('
','*',x'00')
ORDER BY "Total Query Time" DESC
```

If an index was advised for a table scan, it'll be flagged with a 'Y' indicator and the appropriate columns will be listed in the "Columns Advised" field. Obviously there is more information in the 3000 records we could glean, but I leave that as an exercise for the reader.

Furthermore, there are QQRID types that could be joined to our base "non-SQL" query to glean even more information. Like 3001 QQRID, which would show which indexes were used by the non-SQL query.

Again, I'm just giving you a taste of the performance tuning opportunities available at your fingertips by simply using a free, readily available tool on your iSeries – Database Monitor.

Accessing physical file members in SQL

The original idea behind the drive to develop SQL was to supply business analysts and managers with a language for creative and ad hoc database queries. While SQL turned out to be too technical and not even as flexible as regular English grammar, techies ***did*** find it powerful and easy to use. That's how its claim to fame began and today SQL is the standard and dominant method of access to relational database data.

Meanwhile the AS/400 quietly kept fulfilling business needs in its own uniquely adequate manner. One of the unique things AS/400 offered was multi-member support at the file level. SQL designers never thought of members and hence SQL developers were presented with a challenge: how to access data in non-default members present in traditional AS/400 databases.

IBM's solution is ALIAS support whereby a developer can create a permanent object (with object type DDMF, where F stands for File), and access the ALIASed object as a regular file. This is an elegant and permanent solution. Here's a sample:

CREATE ALIAS MYLIB/MYFILEMBR2 FOR MYLIB/MYFILE (MEMBER2)

Traditional OVRDBF (override database file) support is an alternative, but since people needing this support are using SQL, they don't normally opt for it. However, I have seen applications in which member names change dynamically and/or there is a need to create and drop ALIASes rapidly and on-the-fly. Since ALIAS objects are permanent objects, they need to be added to the system's cross-reference tables. If this adding and dropping of ALIASes happens rapidly, it is possible to stress cross-reference OS code to the breaking point.

A good alternative in these dynamic application environments is to fall back to OVRDBF support and its default transient nature. It works well in dynamic environments and can be scoped to *ACTGRPDFN, *CALLLVL and *JOB. However, using the OVRDBF command in SQL is a bit awkward, so I offer a simple stored procedure called **SETMEMBER**. As an SQL developer codes, he can now point his code to the specific member for immediate access. Here is a one-time configuration step of creating a stored procedure:

```
CREATE PROCEDURE QGPL.SETMEMBER (IN member  VARCHAR(10),
                                 IN file    VARCHAR(10),
                                 IN library VARCHAR(10))
LANGUAGE SQL
BEGIN
  DECLARE command CHAR(84);
  DECLARE commandLength DECIMAL(15,5);
  SET command = 'OVRDBF FILE(' || TRIM(member) || ') TOFILE(' ||
                TRIM(library) || '/' || TRIM(file) || ') MBR('
||
                TRIM(member) || ') OVRSCOPE(*JOB)';
  SET commandLength = DECIMAL(LENGTH(TRIM(command)),15,5);
  CALL QSYS/QCMDEXC(command,commandLength);
END
```

And a usage example:

```
      CALL SETMEMBER('member2', 'file', 'library')
            SELECT * FROM MEMBER2
```

The override will disappear along with the job…which is especially useful in ODBC/JDBC environments.

This yet another example of how SQL and traditional RLA (record level access) can coexist peacefully and work together to satisfy diverse and modern business needs.

It also gives old-timers a chance to teach new SQL kiddies a thing or two about how a real system does things.

Useful SQL Scalar Functions
Part 1 - Rounding and Truncating

by Birgitta Hauser

Since the implementation of RPGIV new built-in functions have been added with each Release. Today, including Release V5R4M0, almost 80 built-in-functions are available. Although at first glance that number seems high, keep in mind that SQL provides almost 130 scalar functions, nearly 50% more than RPG does. Even though all SQL scalar functions can be used in RPG with embedded SQL, most programmers either do not know about the scalar functions or how to call them.

In this article you will first learn how to embed SQL statements in RPG and how to use scalar functions. Additionally some mathematical SQL scalar functions will be explained that are either not available or have less functionality in RPG. This article refers only to RPGIV, but all of these scalar functions can be used within any programming language that supports embedded SQL. A review of this information will help keep you from reinventing the wheel.

Embedded SQL

To embed SQL statements into your source code, the member type must be changed to SQLxxxLE, where xxx must be replaced through the actual programming language (i.e. RPG, CBL, C …). Within RPG the SQL statements must be embedded in the C-Specifications and start with C/EXEC SQL and end with C/END-EXEC. Before release V5R4M0 it was not possible to embed SQL statements directly into RPG-free format (even though RPG-free format coding was introduced in release V5R1M0). You either had to use fixed format coding or end the free format coding (with the compiler directive /END-FREE) and after the SQL-Statement restart your free format coding (with the compiler directive /FREE).

Release V5R4M0 is the first release where it is possible to directly embed SQL statements into RPG free format coding, provided the following conditions are fulfilled:

- Each SQL-Statement must begin EXEC SQL.
- Each SQL-Statement must end with a Semi Colon (;), that means END-EXEC must not be specified.
- Each SQL-Statement can be continued over several rows. Continuation rows are not specially labelled, in contrary to fixed format RPG where the continuation rows have to start with C+.
- The code of the SQL-Statement must be placed between position 8 and 80.

(Continued on page 6)

(Continued from page 5)

The following example shows embedded SQL-Statements in RPG fixed format and free format coding.

```
 /Free
   If CustId = *Zeros;
 /End-Free
 * SQL statement in RPG fixed format coding
C/EXEC SQL      Select   Sum(Sales) into :TotalSales
C+                 From  SalesTable
C+                 Where DelivDate between :FromDate and :ToDate
C/End-Exec
 /Free
   Else;
    //SQL statement in RPG free format coding (Release V5R4M0)
      Exec SQL  Select   Sum(Sales) into :TotalSalesCust
                 From  SalesTable
                 Where     DelivDate between :FromDate and :ToDate
                       And CustId = :CustId;
   EndIf;
 /End-Free
```

Example 01: Embedded SQL statements in RPG fixed format and free format coding

Any variable that is defined in either F-, D- or C-Specifications can be used in SQL statements, and must only be labeled with a preceding colon (:). Variables used in SQL statements are also called host variables.

In release V5R3M0 host variables must be unique in the source member. That means defining the same variable locally in several (sub) procedures causes a compile error. With release V5R4M0, the same variable with the same definition in several (sub) procedures will be accepted. Defining a variable with the same name and identical data type but different length in several (sub) procedures will cause a compile error with a severity of 11. Because modules and (service) programs will be compiled correctly up to generation level (GENLVL) 30, the generation level simply can be changed to 11 or higher to get your module or (service) program generated.

SQL Statement SET

Normally SQL is used to receive or change data stored in tables (or physical files). A less known fact is that one can exploit SQL scalar functions without accessing any files whatsoever. SQL/PSM (Persistent Stored Module or SQL programming language) provides the SET statement to set or change the content of variables. This statement can be compared with the EVAL operation code in RPG. Like any other SQL statement, the SET statement can be used in embedded SQL, as shown in the following example.

```
C/EXEC SQL   Set :MyNumer = :MyNumer * 10
C/END-EXEC
C/EXEC SQL   Set :MyAlpha = :FirstName concat :LastName
C/END-EXEC
```

Example 02: SQL SET-Statement

Before writing new functions and reinventing the wheel, or grappling with C-functions (where often pointers must be used) or CEE-APIs, first determine if there is an SQL function that already provides what you want.

Truncating and Rounding

RPG and Cobol are both commercial programming languages, where it is very important that all numeric results are exactly calculated and rounded to a specified number of decimal positions.

In RPG there are two alternatives to handle the odd decimal positions. They either can be ignored (truncated) or they can be rounded commercially or half adjusted. Commercial rounding means: if the digit at the first dropped decimal position is not greater than 4, then it will be truncated; if the digit is greater or equal to 5, then it will be rounded up to the next not dropped decimal position. This guideline is also valid for negative numbers.

(Continued on page 7)

(Continued from page 6)
There is also a mathematical rounding method. Contrary to the commercial rounding where only the first dropped decimal position is considered; mathematical rounding considers all decimal positions. If we round 1.4745 commercially to 2 decimal positions, the result will be 1.47, because the third decimal position is 4. If we round it mathematically the result will be 1.48 because the fourth decimal position is 5 and therefore rounds up the third decimal position to 5 and 5 will be rounded up as shown in the table below.

Comparison Commercial / Mathematical Rounding		
Value	**Commercial**	**Mathematical**
1.4745	1.474 → 1.47	1.4745 → 1.475 → 1.48

Table 01: Comparison Commercial / Mathematical Rounding

This article will only focus on commercial rounding, i.e. the term “rounding” will mean commercial rounding.

Commercial rounding within RPG can be achieved by either adding the (H) extender to the operation code EVAL or by using one of the built-in functions that round half adjusted, such as %INTH() or %DECH(). %INTH() rounds up or down to the next integer value, while with %DECH() the number of decimal positions to be rounded to must be specified.

Now, which scalar functions does SQL provide?

TRUNCATE OR TRUNC – TRUNCATE TO A SPECIFIED NUMBER OF DECIMAL POSITIONS

The scalar function TRUNCATE (or TRUNC) can be compared with the RPG built-in-function %DEC(). As in RPG, the first parameter must contain a numeric value, a numeric variable, or any expression that returns a numeric value. Beginning with release V5R3M0, character variables or character expressions containing digits are accepted as well. When passing a character string, it will be converted into a double precision floating point and then truncated. Additional formatting or casting of the result may be necessary. The second parameter must contain the number of decimal positions the result must have.

The following example shows how the SQL scalar function TRUNCATE can be used within embedded SQL:

```
 * myResult = 2.9999
C/EXEC SQL Set :myResult = Truncate(2.99999999 , 4)
C/END-EXEC

* myResult = -3.999
C/EXEC SQL Set :myResult = Cast(Trunc('  -3.999999 ', 3) as Dec(7, 4))
C/END-EXEC

* myResult = 5
C/EXEC SQL Set :myResult = Truncate(5.12345 , 0)
C/END-EXEC
```

Example 03: Scalar Function TRUNCATE

Contrary to the RPG built-in function %DEC(), the SQL scalar function TRUNCATE provides an additional function. By specifying a negative value in the second parameter the result can be truncated on the left hand of the decimal sign, i.e. –2 means truncating to 100 and –3 means truncating to 1,000.

To achieve the same result with RPG native functions, you first have to divide the original value through 100 or 1,000 (or whatever you need). The result must then be truncated to zero decimal positions using one of the built-in functions %INT() or %DEC(). Finally, the truncated result must be multiplied by the same value that was used to divide the original value.

The following example shows how to truncate to 100 with RPG native functions.

```
/Free
 //MyResult = 12,300
   MyResult = %Int(12334.999 / 100) * 100;
/End-Free
```

Example 04: Truncating to 100 with native RPG functions

(Continued on page 8)

(Continued from page 7)

Using the SQL scalar function TRUNCATE simplifies this calculation, as you can see in the next example.

```
 * myResult = 12,300
C/EXEC SQL Set :myResult = Truncate('     12334.999' , -2)
C/END-EXEC

 /Free
   // myResult = -222,000
   EXEC SQL Set :myResult = Trunc(-222999 , -3);
 /End-Free
```

Example 05: SQL Scalar Function TRUNCATE to truncate to 100 and 1,000

ROUND – COMMERCIAL ROUNDING

To round commercially, SQL provides the scalar function ROUND, which requires two parameters.

In the first parameter any numeric value, variable, character string, or variable containing digits can be passed. In the second parameter the number of decimal positions to round to must be specified. Like the scalar function TRUNCATE, the second parameter accepts negative numbers to round to 10 (for –1), to 100 (for –2), to 1,000 for (-3) and so on. This functionality is not available in the RPG function %DECH(), either. To achieve the same result with native RPG functions additional multiplication and division is necessary.

The following example shows how the SQL scalar function ROUND can be used.

```
 * myResult = 8,743.7
C/EXEC SQL Set :myResult = Round(8743.748 , 1)
C/END-EXEC

 * myResult = 8,744
C/EXEC SQL Set :myResult = Round(8743.748 , 0)
C/END-EXEC

 * myResult = 8,700
C/EXEC SQL Set :myResult = Round(8743.748 , -2)
C/END-EXEC

 * myResult = 10,000
C/EXEC SQL Set :myResult = Round(8743.748 , -4)
C/END-EXEC
```

Example 06: Scalar Function ROUND

SPECIAL ROUNDING

Sometimes it is not enough to round to integers or decimal positions, as rounding to complete 50 or 25 cents is often required. Unfortunately neither RPG nor SQL provide functions that exactly match this need. But the trick we used within native RPG to round or truncate to 100 or 1,000 can be used and modified slightly. For example, when rounding to 100 we first divided through 100, rounded the result to zero decimal positions and multiplied the rounded result with 100. Rounding to 50 cents first requires a multiplication with 2 (in other words a division through one half). This result must be rounded and finally the rounded result must be divided through 2 (or multiplied by one half). It is not mandatory to always round to zero decimal positions. Depending on the requirements, the second parameter in the SQL scalar function can be set to any positive or negative value.

(Continued on page 9)

(Continued from page 8)

The next example shows how to round to 50 cents or 25 cents.

```
 * Rounding to 50 Cent
C/EXEC SQL  Set :myResult = Round((:FirstValue / :SecondValue * 2),  0) / 2
C/End-Exec

 * Rounding to 25 Dollar
C/EXEC SQL  Set :myResult = Round((:FirstValue / :SecondValue * 4), -2) / 4
C/END-EXEC
```

Example 07: Special rounding

FLOOR – ROUNDING DOWN TO THE NEXT INTEGER VALUE

The SQL scalar function FLOOR determines the next integer value that is either equal or smaller than the specified argument. All numeric values, variables and expressions as well as character strings and character variables containing digits (release V5R3M0 or higher) are accepted as input arguments.

As long as only positive values are used, scalar functions TRUNCATE and FLOOR will return the same results. With negative values the results differ. Scalar function TRUNCATE cuts decimal positions so the result may be greater than the original value in some cases. FLOOR on the other hand rounds down to the next integer.

TRUNC(3,75 , 0) Result = 3
TRUNC(-2,5 , 0) Result = -2
FLOOR(3,75) Result = 3
FLOOR(-2,5) Result = -3

Here are some examples that show how the SQL scalar function FLOOR works:

```
* myResult = 5.000
C/EXEC SQL Set :myResult = Floor(5)
C/END-EXEC

 * myResult = 9.000000000
C/EXEC SQL Set :myResult = Floor(9.999999999)
C/END-EXEC

 * myResult = -2.0
C/EXEC SQL Set :myResult = Floor(-1.1)
C/END-EXEC
```

Example 08: SQL Scalar Function FLOOR

RPG does not deliver a comparable built-in-function. To achieve the same result, you first have to verify the passed value is an integer. Integer values stay unchanged. Values with decimal positions must be truncated by using one of the built-in-functions %INT() or %DEC(). After truncating the decimal positions, negative values must be decreased by 1 because cutting off the decimal positions of a negative number will result in the next integer that is greater than the original value.

(Continued on page 10)

(Continued from page 9)

In the following example you'll see the RPG code that is necessary to get the same result as using the SQL scalar function FLOOR:

```
D myResult        S             15P 9
D numValue        S             15P 9
 *----------------------------------------------------------------
 /Free
   If  numValue  = %Int(numValue));
       myResult  = numValue;
   ElseIf numValue < *Zeros;
       myResult  = %Int(numValue) - 1;
   Else;
       myResult  = %Int(numValue);
   EndIf;
 /End-Free
```

Example 09: RPG code – Rounding down to the next integer

Using the SQL scalar function FLOOR instead is much easier as you can see in the following example:

```
D myResult        S             15P 9
D numValue        S             15P 9
 *----------------------------------------------------------------
 /Free
   Exec SQL  Set :myResult = Floor(:numValue);
 /End-Free
```

Example 10: Replacing the RPG code through the SQL scalar function FLOOR

CEILING or CEIL – rounding up to the next integer value

The SQL scalar function CEILING or CEIL is the counterpart to the scalar function FLOOR, i.e. instead of rounding down to the next integer this function rounds up to the next integer.

The following example shows how the SQL scalar function CEILING works:

```
* myResult = 5.000
C/EXEC SQL Set :myResult = Ceiling(5)
C/END-EXEC

 * myResult = 10.000000000
C/EXEC SQL Set :myResult = Ceil(9,000000001)
C/END-EXEC

 * myResult = -1.0
C/EXEC SQL Set :myResult = Ceil(-1,99)
C/END-EXEC

C/EXEC SQL Set :myResult = Ceiling(:HostVar)
C/END-EXEC
```

Example 11: SQL scalar function CEILING or CEIL

There is no RPG built-in-function that's functionally equivalent to CEILING SQL scalar function. The RPG source code to achieve this functionality is similar to the source code for the FLOOR functionality. The only difference is, instead of decreasing the negative values; the positive truncated values with decimal positions must be increased by 1.

Unfortunately neither FLOOR nor CEILING delivers the possibility to round up or down to specified decimal positions or to complete 100 or 1,000. And there are no other functions that deliver these functionalities directly.

(Continued on page 11)

(Continued from page 10)

If you need such functionality, additional multiplications and divisions are required, as shown before in RPG examples.
In the last example you'll learn how to round down to complete 100 and to round up to cents.

```
D myResult        S             15P 5
D myNumValue1     S             15P 5  inz(499.99999)
D myNumValue2     S             15P 5  inz(7.00001)
 *---------------------------------------------------------------------
 /Free
   //Rounding down to 100
     EXEC SQL Set :myResult = Floor(:myNumValue1 / 100) * 100;
     Dsply myResult;

   //Rounding up to Cents
     EXEC SQL Set :myResult = Ceiling(:myNumValue2 * 100) / 100;
     Dsply myResult;

   *InLR = *On;
 /End-Free
```

Example 12: Rounding down to 100 / Rounding up to Cents

But what if we have to always round down to complete 50 cents or round up to 25 cents?
We can use SQL scalar functions FLOOR or CEILING with additional multiplications and divisions as we demonstrated earlier.

The following example shows how to use the scalar functions FLOOR and CEILING to round up and down to complete 25 cents or 50 dollar.

```
D myResult        S             15P 5
D myNumValue1     S             15P 5  inz(1.2345)
D myNumValue2     S             15P 5  inz(50.00001)
 *---------------------------------------------------------------------
 /Free
   //Always rounding down to the next 25 cent
     EXEC SQL Set :myResult = Floor(:myNumValue1 * 4) / 4;
     Dsply myResult;

   //Always rounding up to the next 50 dollar
     EXEC SQL Set :myResult = Ceiling(:myNumValue1 * 0.02) / 0.02;
     Dsply myResult;


   *InLR = *On;
 /End-Free
```

Example 13: Special use of the scalar functions FLOOR and CEILING

This was a short overview over the SQL Scalar Functions for rounding and truncating and how they can be exploited from within RPG.

And now … have fun trying it out!

Thank you Birgitta!

This article is available in German. http://www.centerfieldtechnology.com/pdf/Nuetzliche_SQL.pdf