Falling Off the Cliff

Have you ever experienced the situation where a process that used to take 45 minutes suddenly started to take four or five hours? Join queries are often the root cause behind this type of problem.

When you use SQL to join tables together, special care must be taken to properly tune them. Unlike many types of queries, a poor implementation can result in queries that may take hours (or days!) to complete.

Why are join queries susceptible to the "falling off the cliff" problem? The answer is in the mathematics underlying how joins work. Let's take two tables for example, that contain one millions rows each. If each row in one table was joined to another, the result would be an answer set of 1 trillion rows. The work to produce a trillion rows takes a long time even on powerful computers!

Before we get into the join optimization, here is the terminology used with these types of queries. The table chosen by the DB2 query optimizer to be the root table that all the other tables are joined to is called the **primary table**. This may or may not be the table first specified in your SQL statement. The tables that are referenced by the primary table are called **secondary tables**. The position each table takes in the join order is called its **dial position**. The primary table assumes dial one, the second table is dial two, and so on.

Let's take a more realistic example of two tables with 100,000 and 1,000,000 rows respectively. On average, the first table would likely join to 10 rows in the second table assuming a normal inner join was used and the data was distributed evenly (not likely but for the purposes of this example we'll use that assumption). If an SQL statement joins these two tables together and selects 10,000 rows from the smaller table, the query will return 100,000 rows (10,000 * 10 rows each from the secondary table). To do this using a traditional nested loop join, each of the 10,000 rows selected from the primary table will be used to build a key, and that key will be used to probe an index that contains the columns in the secondary table that relate the two tables. This process uses the following system resources:

- CPU to build the key from the selected row.
- CPU to search the index over the secondary table.
- Disk I/O when the index is probed.
- Disk I/O when the row associated with the key is referenced
- CPU to map the secondary table's row into a buffer

As can be seen, if this process is done unnecessarily, a join query can run for a very long time. As secondary tables are added to the query the cost of incorrect join order can drastically impact performance because the number of potentially unnecessary operations increases very, very quickly.

Let's take our simple example again. If the query optimizer reverses the join order, the table with 1,000,000 rows is now processed first. Each of these rows is used to construct a key, do an index probe, and then selection is performed on the secondary table. Even though the same number of rows is returned by the query, the internal processing requires 1,000,000 * 100,000 rows to be processed before selection reduces that number to the 100,000 join rows. This seemly minor mistake results in approximately 1,000,000 times more work than the first query!

Determining the correct join order is a very complex problem for the DB2 query optimizer. Consider the case where an SQL statement joins four tables, the query optimizer can examine 12 or more table orders. As the number of tables and join relationships expand, the likelihood of potential errors expands – as does the consequence of a poor choice.

In our next newsletter, we will go into depth on the options you have to help the query optimizer make the right choices and keep your applications from "falling off a cliff".

the cliff

i want it fast. i want it now.

Old Gotcha
with a New Twist

In this article I am going to discuss the perennial gotcha pertaining to the use of variable length (VARLEN) fields as well as the latest twist in this area, so you old-time AS/400 aficionados should stick around for this one as well.

As part of Centerfield's database/ ASSESSMENT service, we often talk to customers who use variable length fields. Folks with long-time AS/400 experience are usually familiar with the way variable length fields work and the best allocations for their database fields. Many times though, we'll talk to a customer who has either ported their SQL application from a different platform or was simply taught SQL design basics on a different database (SQL Server, Oracle etc). These customers are not aware of the nuances involved in the use of variable length columns in their database.

VARCHAR advantages

The main advantage of using variable length columns (VARCHAR or VARGRAPHIC) is that they are easy ways to reduce the table size residing on disk. Database disk consumption is an important consideration on many database platforms, to the point where Database administrators use complex formulas to properly gage their database size requirements and implement appropriate actions (purchase more disks, increase database or table size limits, etc). Due to the scalable nature of DB2 for iSeries, we don't really dwell too much on this issue but even on our platform we know that a lot of character data varies in length dramatically (think of executive memos) so some space on disk is really white space. Variable length column design is simple and elegant; it is basically a structure

where the leading two bytes hold the field's length and the rest of the buffer is actual user data. For these two reasons (disk space saving & ease of use), VARCHAR or VARGRAPHIC are widely used with SQL & RLA (Record Level Access) developers.

First gotcha

VARLEN field data is stored internally in two different areas: ALLOCATE & OVERFLOW.
The designer (you!) controls how much data is stored in each area. The tradeoff is clear and rather mundane: disk space versus performance. If your principal design objective is to save on disk

Can't Wait for IBM?

Sometimes you need certain system functionality and can't find it. You talk to IBM and perhaps they have it in a development plan for the next release or more likely they suggest you open a DCR (design change request) and start the process of getting the function into the operating system.

What if you really, really need it NOW? Chances are you can purchase a tool, write your own function, use some MI interface, etc. Every once in a while though, a function you need can only be provided by the OS developers. One example of such a function requirement is the programmatic ability to retrieve current values of some system configuration commands before you actually change them, or execute some related function that depends on a particular system setting.

A case in point is the CHGFTPA command with which you can easily check to see if SSL is enabled by prompting the CHGFTPA command. But how would you do it within your program? There is no DSPFTPA to OUTFILE or RTVFTPA command. There is no RetrieveFtpAttributes API either. Perhaps you can dump a system space object and parse the spooled file? Sometimes that works if the space object is accessible and easy to parse.

I thought about this and believe that I have found an easy alternative to this specific issue. A lot of these configuration commands like CHGFTPA or CHGQRYA have prompt override programs (POPs) that retrieve the values when you prompt the command and POP data is exactly what you're after. The challenge is that these POPs are created in *SYSTEM state, and as such cannot be called by user programs. However, there is a little known Retrieve Prompt Override API (QPTRTVPO) that allows us to get at exactly what we want. Here's a link describing the API:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/apis/qptrtvpo.htm

The basic design concept is to create a RTV* type command with a command processor program (CPP) that utilizes the QPTRTVPO API to retrieve requested attributes and pass them back to the invoking program. The complexities of parsing the API output would be hidden by the CPP and the CL programmer would simply interface with the well-known RTV* type command interface, ala RTVJOBA. The nice thing about RTV* commands is that the programmer doesn't have to declare all the parameters…just the ones actually required by the program.

They say a picture is worth a thousand words. Similarly, in programming, a simple code sample is worth a ton of articles so I'm not going to leave you hanging:

http://www.centerfieldtechnology.com/files/RTVFTPA source code.zip

If you face this type of requirement and can't wait for IBM, I hope you find this simple technique a plausible alternative.

Fun reading.

Elvis
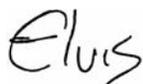
consumption you can go to the extreme and specify ALLOCATE(0) when you create the column, i.e.:

CREATE TABLE QGPL/T1 (C1 VARCHAR (200) ALLOCATE(0))

Then all of your data will go into the OVERFLOW area, but **only** your data and nothing else. This means that you will not consume any unnecessary space in the ALLOCATE area which is the normal part of the file's fixed length data space. The problem with this is that your performance now may suffer, as every time you touch this data, the database code needs to reach into the OVERFLOW area. This does not perform nearly as well as fetching data from the ALLOCATE area, as two trips to the disk are required: one to read the fixed portion of data and other to read OVERFLOW area data thus doubling the work.

The first gotcha for folks groomed on other platforms is that the iSeries **default** value for the ALLOCATE parameter is 0, so if we omit it in our CREATE TABLE statement, that's what we'd get. This is not how other database platforms behave and is often one of the culprits that make ported SQL applications perform poorly on the iSeries when initially rolled out. Old-time AS/400 aficionados know very well that you have to do some performance tuning every time you roll-out new SQL apps or modules, but will the uninitiated new-comer know that as well? And how long will it take him/her to learn this slight difference in the way VARCHARs and VARGRAPHICs behave?

Tradeoff

We know disk is getting cheaper all the time and performance is often paramount, so my bias is toward performance. The magic ratio you should follow is to ensure your ALLOCATE value encompasses **at least** 90-95% of all of your rows. This will ensure good performance in most cases. For the occasional odd duck, making an extra trip to the disk is an acceptable tradeoff.

Let's say you want 99% of your rows to fit in the fixed area and only 1% in the overflow area (a common scenario). Here are couple of SQL statements you can run over your table to figure the right value to use in the ALLOCATE field:

First, let's figure out how many rows comprise 1%:

*SELECT COUNT(*) * 1/100 AS OnePercent FROM qgpl/t1*

Now let's see your row length groupings and how many rows are in each grouping:

SELECT   LENGTH(RTRIM(c1)) FieldLength,
*   COUNT(*) RowCount*
FROM   qgpl/t1
GROUP BY LENGTH(RTRIM(c1))
ORDER BY 1 DESC

Add enough rows from the top to get you above the 1% threshold and then use the last-added field's length value

USEFUL SQL SCALAR FUNCTIONS
PART 2 - DATE AND TIME FUNCTIONS

by Birgitta Hauser

Date and time calculations are and always will be a hot issue. In release V5R1 RPG was enhanced with several new built-in functions for date and time calculation. Some of these new functions were only replacements for operation codes that are not supported in RPG free format coding. In the subsequent releases no further new built-in-functions for date and time calculation were added.

Contrary to SQL! With each of the previous releases SQL was enhanced with several powerful and important scalar functions for date and time calculation. For most of these functions RPG has neither an equivalent nor something comparable. Sure, in RPG you can piece together whatever you need, but why not simply exploit the existing SQL scalar functions? This article will focus on the SQL scalar date and time functions that allow you to determine date and time portions for a given date, time or timestamp.

Calculating Date or Time Portions

Both RPG and SQL provide a set of built-in scalar functions that allow you to determine the day number, the numeric month or the year for a given date or timestamp. Analogue functions exist to extract the hour, minute or second portion from a given time or timestamp.

RPG delivers the operation code EXTRCT as well as the built-in function %SUBDT(). By the way, %SUBDT() stands for substring date/time, not for subtract. For both operation code and the built-in function, a date or time unit must be specified, i.e. *YEARS or *Y for years, *MONTHS or *M for months, *DAYS or *D for days, *HOURS or *H for hours, *MINUTES or *MN for Minutes, *SECONDS or *S for seconds and *MSECONDS or *MS for microseconds.

Contrary to RPG, SQL provides a specific scalar function for each date or time unit as listed below:

YEAR	Extracts the year part from a given date or timestamp or from a date or timestamp duration.
MONTH	Extracts the month part from a given date or timestamp or from a date or timestamp duration.
DAY	Extracts the day of months from a given date or timestamp or from a date or timestamp duration. Not to confuse with the scalar function DAYS that calculates the number of days since 01/01/0001.
DAYOFMONTH	Extracts the day of months from a given date or timestamp like the scalar function DAY. But contrary to the scalar function DAY, it cannot be used to determine the day part of a date or timestamp duration.
HOUR	Extracts the hour part from a given time or timestamp or from a time or timestamp duration.
MINUTE	Extracts the minute part from a given time or timestamp or from a time or timestamp duration.
SECOND	Extracts the second part from a given time or timestamp or from a time or timestamp duration.
MICROSECOND	Extracts the microsecond part from a given timestamp or a timestamp duration

Depending on which function is used, either a date, time or timestamp; or their characters' representations, can be passed as parameter. It is also possible to pass a numeric date, time or timestamp duration.

Reader Input

I read your article in the "Out in Left Field" newsletter about case-insensitive searches on iSeries. The *LangIDShr sort-sequence has been absolutely invaluable to our shop for several years.

One drawback to that method is that IBM will not support it in SQE until at least V5R5. If you use the *LangIDShr sort-sequence, you're essentially BLOCKED from using some of the newer SQL constructs (since some of them ONLY run under SQE).

Best Regards,
DF, Senior Developer / Analyst

I just wanted to compliment you and the staff at Centerfield for what I believe to be the most useful newsletter I've ever received.

The tips contain great content, spiced with just the right amount of humor, and formatted for easy reading and/or printing, if you are on the go.

Thank you for all you do for us. I certainly hope we will have the opportunity to do business with Centerfield Technology at some point in the future!

Best regards,
DD, Systems Analyst

Aw…shucks...thanks!!

Welcome partners!

Centerfield Technology formed two new partnerships this summer.

In July, Centerfield signed an exclusive agreement with Sirius Computer Solutions a dominant System i hardware and application reseller. Centerfield Technology will assist Sirius in maximizing customers' iSeries database performance utilizing tools, training and consulting.

Stan Staszak Director - iSeries Product:

"Centerfield's performance monitor/tuning and DASD management tools are the best in the industry and should help our customers greatly improve performance and productivity in their iSeries/System i environments. I think this is particularly exciting because some of Centerfield's tools are specifically designed for JDE EnterpriseOne/OneWorld applications...and a good majority of our customers are running JDE products."

On a different front, a marketing partnership with Innovatum will leverage the synergism between the two companies, which already share many customers and prospects.

Ardi Batmanghelidj, President, Innovatum, Inc.:

"We will work hand-in-hand with Centerfield as we expand our presence in the burgeoning market for iSeries compliance and auditing tools, particularly in the financial and pharmaceutical industries. With this partnership we look forward to bringing our existing customers, many of whom run high transaction environments, the benefit of CTI's fantastic SQL performance monitoring/tuning tools."

(Old Gotcha with a New Twist —Continued from page 3)

as the ALLOCATE value. You can use the ALTER TABLE command to alter an existing field's ALLOCATE value, i.e.:

    ALTER TABLE QGPL/T1 ALTER COLUMN C1 SET DATA TYPE VARCHAR (200) ALLOCATE(50)

Second gotcha

Large Object support was introduced to the iSeries over a decade ago. Today you can define column data types as LOB, BLOB, CLOB, DBCLOB etc. These data types are only available in SQL DDL and are variable length types by design. The second and newer gotcha relating to variable length implementation is: VARCHAR and VARGRAPHIC share the OVERFLOW area with LOBs. This has an unfortunate side effect in that when you touch a VARCHAR field that requires reaching into the OVERFLOW area, any LOBs in that same row will be paged in as well. These could be many megabytes or even gigabytes in size! And you might not even need the LOB column!

This strengthens the argument I made for biasing strongly toward performance and making the ALLOCATE keyword as large as 99%, which would ensure less trips to the OVERFLOW area and thus reduce potential paging of LOBs into main memory.

Other tips

If you're doing any programming against tables with variable length columns and are NOT using variable length host variables, please start using them NOW. Using fixed length host variables is likely to cause unnecessary spillage into the overflow area.

I realize that in shops running 24/7 it may be hard to schedule a re-org of a table (RGZPFM), but if at all possible, do it periodically. With the new re-org-while-active support IBM has introduced in V5R3, it may be a lot easier to push this request through. Besides removing deleted records out of a table, RGZPFM has the secondary benefit of compacting the fragments that are not in use in the OVERFLOW area. This reduces the read time for rows that overflow, increases the locality of reference, and produces an optimal order for serial batch processing. In other words, it benefits performance and that's the name of the game.

Use the DSPFFD command to view current allocations for variable length fields and pay special attention to ones listing "0" or "None."

Good luck and have fun!

Elvis

Elvis Budimlic
Development Director

5

Valid character representations for time are:
- HH.MM.SS   International Standards Organization (ISO) format or IBM European standard (EUR) format where HH represents hours, MM minutes and SS seconds.
- HH:MM:SS   Japanese Industrial Standard Christian era (JIS)
- HH:MM AM (PM)   IBM USA standard format (USA)

For all character representations of time formats except the USA format, the hour part can be any value between 0 and 24. For USA representation, the hours must be specified between 1 and 12 with the exception of 0:00 AM.

If you pass a USA character representation of a time and use the scalar function HOUR to determine the hour part of this time, the result will be a value between 0 and 23. In the following example the hour part of '3:30 PM' will be computed. The result will be 15, not 3.

```
Select  HOUR('3:30 PM')
   From SysIBM/SysDummy1;
```
Example 1: Extracting the hour part from a given character representation of a time

Time duration represents numbers of hours, minutes and seconds defined as a DECIMAL(6,0) number. To be properly interpreted, the number must have the format HHMMSS. Even if you use the USA time format in your job, the hours must be specified between 0 and 23.

Valid character representations for dates are:
- YYYY-MM-DD   ISO or Japanese Industrial Standard Christian era (JIS)
                   Where YYYY represents the Year, MM the month and DD the day part
- DD.MM.YYYY   IBM European Standard format (EUR)
- MM/DD/YYYY   IBM USA Standard format (USA)

For character representation, date and time separators are predefined and cannot be changed. SQL is able to identify the valid date even if the day and/or the month are specified as single digit only, as long as the year has four digits and the correct date separator is used. A date with only a two-digit year cannot be identified as a valid date, regardless of which date separator is used.

Date duration represents number of years, months, and days expressed as a DECIMAL(8,0) number. Regardless of what date format is used in your job, the number must represent the format YYYYMMDD to be correctly interpreted.

Valid character representations for timestamps are:
- YYYY-MM-DD-HH.MM.SS.MSMSMS   IBM-SQL-Standard
- YYYYMMDDHHMMSS   A character representation that consists of 14 digits
- YYYY-MM-DD HH:MM:SS.MSMSMS   ISO-Format (new with release V5R4)

Timestamp duration represents a number of years, months, days, hours, minutes, seconds, and microseconds, expressed as a DECIMAL(20,6) number. To be properly interpreted, the timestamp duration must be specified as YYYYMMDDHHMMSS,MSMSMS.

The SQL scalar functions DATE, TIME or TIMESTAMP that convert the character representation into a real date, time or timestamp must only be used if you need to perform computations with the results. For example, if you want to calculate date or time differences between these dates or times.

The following example demonstrates how to use these SQL scalar functions. It is the SQL script for a User Defined Function (UDF) to convert a real date into a numeric value with the format YYYYMMDD. This function may be helpful if you have to work with legacy applications where all date and time expressions are stored as numeric values.

```
CREATE FUNCTION MySchema/CvtDateToNum (ParDate Date)
        RETURNS DECIMAL(8, 0)
        LANGUAGE SQL
        NOT DETERMINISTIC
        CONTAINS SQL
        CALLED ON NULL INPUT
        DISALLOW PARALLEL


 RETURN Cast(Year(ParDate)  * 10000 +
             Month(ParDate) *   100 +
             Day(ParDate)
         as Dec(8, 0 ));
```
Example 2: UDF CvtDateToNum → Conversion of a real date into a numeric representation

The following example shows how to use UDF CvtDateToNum in a SQL statement. In the following statement, the Best Before numeric date will be updated with the numeric date value 10 days in the future for all rows except where Production Date is set to today.

```
Update MySchema/MyTable
   Set BB4Dat = CvtDateToNum(Current_Date + 10 Days)
 Where PrdDat < CvtDateToNum(Current_Date);
```
Example 3: How to use the UDF CvtDateToNum in an SQL statement

The conversion from dates into numeric values as well as the inverse is one of the few features that RPG provides while SQL has no equivalent. Instead of creating a SQL defined UDF, a small RPG function could be written, that consists of a single RETURN statement using the built-in function %DEC(MyDate: *DateFormat) to convert a date into a numeric value. This RPG function could be registered as external UDF and used as any other scalar or user defined function.

With release V5R3 the scalar function EXTRACT which can be considered as equivalent to the RPG operation code EXTRCT resp. the built-in-function %SUBDT() was introduced. When using the scalar function EXTRACT you must specify the time unit you want (i.e. YEAR, MONTH, DAY, HOUR, MINUTE, SECOND) in the first parameter. The second parameter requires a date, time, timestamp or a valid character representation of a date, time or timestamp. The result returned by the scalar function EXTRACT corresponds to the results returned by the single scalar functions such as YEAR, MINUTE etc., with the following exception:

- The result for seconds will be returned as DECIMAL(8, 6) numeric value, where the decimal positions represent the microseconds.
- Microseconds cannot be specified as a time portion and must be returned with seconds.

In the following example we use the scalar function EXTRACT to determine seconds and microseconds parts for the current timestamp.

```
D MyTimeStamp     S                 Z    inz(*SYS)

D DSSecMs         DS
D   SecMS                         8S 6
D    Sec                          2S 0  Overlay(SecMs)
D    MS                           6S 0  Overlay(SecMs: *Next)
 *-------------------------------------------------------------------
C/EXEC SQL  :SecMS = Extract(SECOND from :MyTimeStamp)
C/END-EXEC
```
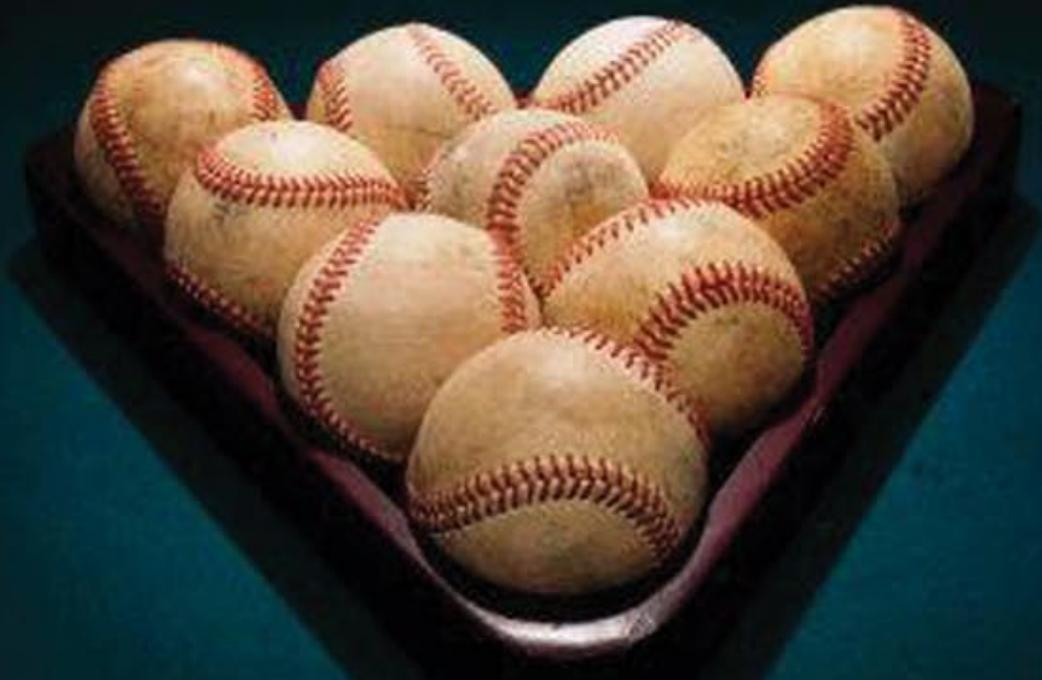*Example 4: SQL Scalar Function EXTRACT*

DAYOFYEAR – Calculating the day of year

Sometimes a numeric day of year must be calculated for a given date or timestamp. There is no function that will return this result directly in RPG. The only way is to calculate the difference in days between a given date or timestamp and December 31st of the previous year. Alternatively, the difference in days between a given date and January 1st of the same year can be calculated and one day added. In either method several RPG statements or a rather complex formula would be needed.

SQL provides scalar function DAYOFYEAR that returns a numeric value between 1 and 366 for a given date or timestamp or a valid character representation of a date or timestamp. Number 1 represents January 1st and 365 or 366 represent December, 31st, depending on whether the year is a leap year or not.

The following example shows how scalar function DAYOFYEAR can be used in RPG with different date and timestamp values.

```
D DateIso         S                  D   inz(D'2006-12-31')
D TimeStampIso    S                  Z   inz(*Sys)
D DateAlpha       S              10A  inz('1.6.2006')
D TimeStampAlpha  S              26A  inz('2006-7-1-8.30.23')
D DayYear         S               3P 0
 *-------------------------------------------------------------------------
C/EXEC SQL   Set Option DatFmt = *ISO
C/END-EXEC

C/EXEC SQL   Set :DayYear  =  DayOfYear(:DateIso)
C/END-EXEC

C/EXEC SQL   Set :DayYear  =  DayOfYear(:TimeStampIso)
C/END-EXEC

C/EXEC SQL   Set :DayYear  =  DayOfYear(:DateAlpha)
C/END-EXEC

C/EXEC SQL   Set :DayYear  =  DayOfYear(:TimeStampAlpha)
C/END-EXEC
C                eval       TimeStampAlpha = '20060528173025'
C/EXEC SQL   Set :DayYear  =  DayOfYear(:TimeStampAlpha)
C/END-EXEC
```

Example 5: How to profit from the SQL scalar function DAYOFYEAR in RPG

DAYOFWEEK and DAYOFWEEK_ISO – Calculating the Day Of Week

Some of the most important scalar functions with no existing equivalents in RPG are DAYOFWEEK and DAYOFWEEK_ISO. Both functions return numeric day of week for a given date or timestamp or a valid character representation of a date or timestamp. The difference between both functions is the establishment of the first day of week.

When using the scalar function DAYOFWEEK, the Sunday is defined as the first day of week and consequently returned as 1, while 7 is returned for Saturday.

According to the ISO guidelines however a week must begin with the Monday and end with the Sunday. The scalar function DAYOFWEEK_ISO satisfies these requirements, i.e. 1 is returned for Monday and 7 for Sunday.

The next example shows how easy it is to calculate day of week in RPG that complies with ISO guidelines:

```
D DateIso          S                  D   inz(D'2006-05-30')
D TimestampAlpha   S              26A  inz('2005-5-31-10.23.47.123000')
D DayOfWeekNum     S              1P 0
 *-------------------------------------------------------------------------
C/EXEC SQL   Set Option DatFmt = *ISO
C/END-EXEC

C/EXEC SQL   Set :DayOfWeekNum  =  DayOfWeek_ISO(:DateIso)
C/END-EXEC

C/EXEC SQL   Set :DayOfWeekNum  =  DayOfWeek_ISO(:TimeStampAlpha)
C/END-EXEC
```
Example 6: Calculating the numeric day of week according to the ISO guidelines

DAYNAME – Determining Name of the Day of Week (Release V5R3M0)

Finding an easy way to calculate the numeric day of week is already a big improvement. Sometimes this is not enough, particularly when the name of the day of week is needed. Sure, names can be stored in a compile time or runtime array and numeric day of week can be used as an index to return the appropriate name. No magic there, either.

With release V5R3 the scalar function DAYNAME was introduced. This function allows you to determine the name of the day of week for a given date or timestamp or an appropriate character representation. Names for the day of week are stored in the message file QCPFMSG with the message-id CPX9034. In this way concerns about the national language usage can be dissipated. If I use the scalar function on our iSeries (it's not yet System i), with primary language German, German names for the day of week will be returned.

In the following example you'll see how the name of the day of week can easily be determined for a given date or timestamp.

```
D DateIso          S                  D   inz(D'2006-05-30')
D TimestampAlpha   S              26A  inz('2005-5-31-10.23.47.123000')
D DayOfWeekName    S              10A
 *-------------------------------------------------------------------------
C/EXEC SQL   Set Option DatFmt  = *ISO
C/END-EXEC

C/EXEC SQL   Set :DayOfWeekName =  DayName(:DateIso)
C/END-EXEC

C/EXEC SQL   Set :DayOfWeekName =  DayName(:TimeStampAlpha)
C/END-EXEC
```
Example 7: Determining the name of the day of week with the scalar function DAYNAME

MONTHNAME – Determining the Name of Month (Release V5R3M0)

Along with the scalar function DAYNAME, another similar scalar function was added in release V5R3: MONTHNAME. It allows you to determine name of month for a given date, timestamp or a valid character representation of a date or timestamp. Like the day names, month names returned by the scalar function MONTHNAME are stored in the message file QCPFMSG, but with the message-id CPX3BC0.

In the following example the month name for a given date resp., a given timestamp is determined using the scalar function MONTHNAME.

```
D DateIso          S               D    inz(D'2006-05-30')
D TimestampAlpha   S               26A  inz('2005-5-31-10.23.47.123000')
D DayOfWeekName    S               10A
 *--------------------------------------------------------------------
C/EXEC SQL   Set Option DatFmt  = *ISO
C/END-EXEC

C/EXEC SQL   Set :DayOfWeekName =  MonthName(:DateIso)
C/END-EXEC

C/EXEC SQL   Set :DayOfWeekName =  MonthName(:TimeStampAlpha)
C/END-EXEC
```
Example 8: Determining the month name with the scalar function MONTHNAME

To give you an illustration of how several of these scalar functions can be used in combination, the
following example shows the conversion of a given date into a character representation with the
format "Day name, month name, day number, year".

```
D DateIso          S               D    inz(D'2006-07-17')
D DateText         S               50A  varying
*--------------------------------------------------------------------
C/EXEC SQL   Set Option DatFmt  = *ISO
C/END-EXEC

C/EXEC SQL
C+ Set :DateText  = Trim(DayName(MyDate))   concat ', ' concat
C+                  Trim(MonthName(MyDate)) concat ', ' concat
C+                  Trim(Char(day(MyDate))) concat
C+                  case when day(MyDate) in (1, 21, 31) then 'st'
C+                       when day(MyDate) in (2, 22)     then 'nd'
C+                       when day(MyDate) in (3, 23)     then 'rd'
C+                        else 'th' End      concat ' ' concat
C+                  Char(year(MyDate))
C/END-EXEC
```
Example 9: Converting a date into a specified character representation with month and day names

QUARTER – Determining the Quarter
The scalar function QUARTER can be used to determine the quarter for a given date, timestamp or
character representation of a day or timestamp. If the month of the given date or timestamp is
January, February or March, the result will be 1 for the first quarter. Number 2 will be returned for
months April, May and June, and so on.

To compute the quarter with RPG native functions, first the month part for the given date or timestamp
must be determined. Depending on the month, the quarter can be calculated by using several IF or
SELECT-Statements. No magic either, but using the scalar function QUARTER makes life at least a
little easier, as shown in the following example.

```
D MyTimeStamp      S               Z    inz(*SYS)
D MyQuarter        S               5I 0
 *--------------------------------------------------------------------
C/EXEC SQL   Set Option DatFmt  = *ISO
C/END-EXEC

C/Exec SQL   Set :MyQuarter = Quarter(:MyTimeStamp)
C/END-Exec
```
Example 10: Computing the quarter with the SQL scalar function QUARTER

WEEK and WEEK_ISO – Calculating the Week Number

Another very important and often required date unit is the calendar week or week number. For example: "The goods will be delivered in week 23." RPG provides no built-in-function to easily determine week number, contrary to SQL that offers scalar functions WEEK and WEEK_ISO.

The scalar function WEEK returns the week for a given date, timestamp or a valid character representation for a date or timestamp. When using the scalar function WEEK, Sunday is considered as the first day of the week and January 1st is always in the first week.

According to the ISO guidelines ISO 8601 calendar week is defined as follows:
- The calendar week begins always with the Monday
- The first week of a year must include at least 4 days of the new year.
  In other words:
  - January, 4th always belongs to the first week of a year.
  - The first Thursday of a year is always in week number 1.

In this way, December, 31st can be found in the first calendar week of the next year, while in another year, January, 1st belongs to the last week of the previous year. Depending on the date of the first day in the first calendar week and if a year is a leap year or not, a year can have 52 or 53 calendar weeks.

If you have to calculate the week number according the ISO guidelines manually, you'll end up writing a bunch of code. Using the scalar function WEEK_ISO you can reduce your source code to a single SQL statement. In the following example you'll see an RPG function that determines calendar week and year (current, previous, next) in which the week belongs. The year is also calculated with an SQL statement to show again how SQL scalar functions can be used. There is no advantage to using SQL over native RPG commands to compute the year.

```
H NoMain
 *-----------------------------------------------------------------------
 * Prototypes
D/Copy QPROLESRC/DateFkt
 **********************************************************************
P* Procedure name:  YearWeekIso
P* Purpose:         Compute Year and Week Number for a given date
P* Returns:         YYYYWW  (6S 0)
P* Parameter:       ParmDate   => Date
 **********************************************************************
P YearWeekIso     B                      Export
D YearWeekIso     PI             6S 0
D   ParmDate                      D

D DsYearWeek      DS
D   YearWeek                      6S 0
D   DsYear                        4S 0 Overlay(YearWeek)
D   DsWeek                        2S 0 Overlay(YearWeek: *Next)
 *-----------------------------------------------------------------------
C/EXEC SQL   Set Option DatFmt = *ISO
C/END-EXEC

C/EXEC SQL   Set :DSWeek = Week_ISO(:ParmDate)
C/END-EXEC

C/EXEC SQL
C+ Set :DSYear =  Case When Month(:ParmDate) = 12 and :DsWeek  =  1
C+                     Then Year(:ParmDate)  +  1
C+                     When Month(:ParmDate) =  1 and :DSWeek >= 52
C+                     Then Year(:ParmDate)  -  1
C+                     Else Year(:ParmDate)  End
C/END-EXEC
C                    Return    YearWeek
P YearWeekIso     E
```

Example 11 : Calculating the calendar week and the appropriate year for a given date

This was a short overview of SQL scalar functions to determine date and time portions for given dates, timestamps or times.

And now, have fun when trying them out!