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

SQL in CL...What a Variety!

CL (Control Language) is an irreplaceable programming/scripting language on the System i. It integrates seamlessly with System i commands and work management messaging. However, many differences between CL and true HLLs (higher level languages) like RPG, C and COBOL become obvious once CL is used. One of the more common pet peeves is lack of integrated support for file writes, especially in comparison to another easy yet powerful programming language: SQL. That prompts me to take a look at the variety of innovative methods System i folks have chosen to bridge this gap.

Let's work our way up from the most common to least common methods, all of which are unique and powerful:

RUNSQLSTM

The Run SQL Statement command is *the* IBM recommended and supported method of running SQL statements in CL. The command allows for: any non-SELECT statement; multiple statements per source member separated by semicolons; commitment control; system versus SQL naming; etc. The command works very well for large batch processes working on static files with fixed parameters. I use it exclusively for my DDL (Data Definition Language) scripts, building tables, indexes, views, stored procedures and UDFs for many of our databases. RUNSQLSTM**,** however, falls short in error handling and lacks the ability to pass parameters dynamically.

STRQMQRY

Start Query Management Query excels where RUNSQLSTM falls short. It allows for dynamic parameters, has better error messaging and has a development environment started via the STRQM (Start Query Manager) command, which allows for prompting of SQL inside a source member. STRQMQRY allows for SELECT statements, giving output options of display screen (the default), spooled file and output file, and its output can be further massaged using query manager forms.

Let's take a look at a simple example of creating a CL command that runs generic SQL requests:

CRTLIB RUNSQL
CHGCURLIB RUNSQL
CRTSRCPF FILE(RUNSQL/QQMQRYSRC) RCDLEN(79)
STRSEU SRCFILE(RUNSQL/QQMQRYSRC) SRCMBR (RUNSQL) TYPE(SQL) OPTION(2)
type following four characters: *&SQL*
F3
Enter
CRTQMQRY QMQRY(RUNSQL/RUNSQL) SRCFILE (RUNSQL/QQMQRYSRC)

Now that a QM query object taking generic input has been created, we can run arbitrary statements by passing our entire SQL query in as input:

*STRQMQRY QMQRY(runsql/RUNSQL) SETVAR((SQL 'CREATE TABLE T7 AS (SELECT * FROM QSQPTABL) WITH DATA'))*

To make this into a generic CL command all you need is a CL program that takes single input and runs this QM query object passing that SQL statement variable in as a parameter. Finally, create a command object that takes a one parameter 55 byte character variable.

"Why 55?" you ask. "That's not nearly enough for my SQL statements!" Well, that *55 characters limitation on a single parameter* is one of the drawbacks of STRQMQRY. You'd be correct to think this is easily worked around by having multiple QM variables and then passing input parameters as 55 byte chunks, except that then you hit another weirdness…the way QM automatically trims blanks. Do not despair though. A System i techie named Buck Calabro has already worked through these gyrations and has come up with a popular command interface that leverages this technique:

http://faq.midrange.com/data/cache/322.html

To be sure, different authors have leveraged this technique and there a

(Continued on page 6)

Inside this issue:

COVER STORY: SQL in CL...What a Variety!	1
ELAs: Trimming Licensing Costs	2
Parallel Divergence	3
CTO Memo	4
Insurance Policy	4
Delete Old IFS Files	5

Hurry HomeRun upgrade offer expires soon!

ELAs

Trimming Licensing Costs

Many approaches have been used to price and license software. Each of these methods has advantages and disadvantages. For example, per-user pricing has an advantage in that you only pay for the actual number of people using a particular piece of software. The downside is that depending on the vendor's pricing schedule, if demand for the software increases rapidly, you may end up paying a much larger than expected price for additional licenses.

For the large company, the choices become harder because the number of machines, potential for use (or nonuse!), locations, and corporate priorities all interact to make it easy to miscalculate the long-term costs of the software.

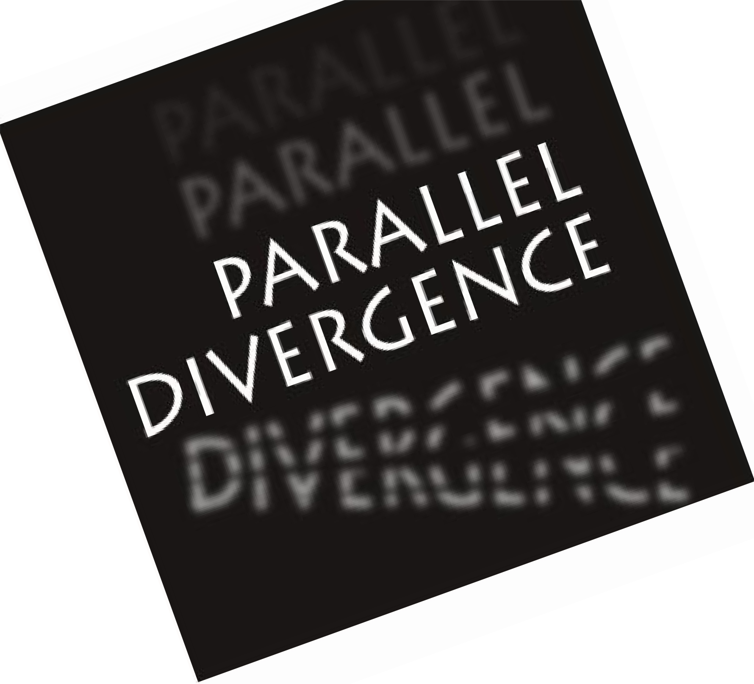
Some companies have taken a different approach to purchasing software. For products that align with their corporate hardware strategy, it makes the most sense to negotiate licenses for the whole company – thus the term "Enterprise License Agreement." While the details will differ from vendor to vendor, the basic idea is that for one price the software can be used on any machine in any location within the company.

There are several important advantages of this approach. The first is pricing. Clearly, large bundles of software can be discounted more aggressively than if the software is purchased in smaller increments over time. Typically multi-year maintenance agreements can be negotiated at the same time to make the longer term cost more predictable as well.

The second benefit is flexibility. Instead of trying to anticipate hardware or software purchases across the enterprise, the license agreement will typically allow the benefits of the package to be garnered whenever and wherever it makes sense. This obviously reduces the risk and can dramatically increase the benefit without forcing the IT department to ask for additional budget or eliminate a project to make room for an unexpected purchase.

Thirdly, from an IT administration standpoint, managing the relationship with the vendor and the software keys becomes a routine task that can be handled by anyone in IT rather than having to be dealt with at the Director, CTO, or CFO level.

Lastly, the group that negotiates an Enterprise License Agreement can become a "hero" to the rest of the organization if they ensure that the software's value gets spread among peer departments or divisions.



As most SQL programmers and database administrators know by now, there are two query engines on the System i. The Classic Query Engine (CQE) is the original database component and was shipped in the early 1990's. CQE not only ran SQL queries but was also used by Open Query File and Query/400. In an aggressive effort to modernize and improve the performance of SQL on the iSeries in the late 1990's, IBM decided to revamp a large portion of the software stack that processes statements. The result, called the SQL Query Engine (SQE), has been slowly phased in over several releases beginning in V5R1 and as of V5R4 is used by a majority of statements. Older interfaces like OPNQRYF and RUNQRY still depend on the CQE.

Theoretically, customers should not actually notice that there are two different query engines on their system. As we all know however, theory is…well, only theory and not real world. With any large and complex piece of software there are bound to be subtle differences ranging from inconsequential to those that are critical to understand.

To see if your queries are using SQE or CQE see a previous newsletter article on this topic at this link: http://www.centerfieldtechnology.com/publications/archive/April%202006.pdf

The increased power of systems and the growth in multi-processor machines has opened up an opportunity to run single SQL statements on more than one CPU. This capability is enabled by the **DB2 for i5/OS SMP feature**. More and more shops are purchasing this feature with the hope of cutting their SQL processing time down by taking advantage of *multiple* powerful processors. There are several ways to take advantage of the parallel features of SQL:

- Change the QQRYDEGREE system value (affects all jobs)
- Use the CHGQRYA command (affects one job)
- Use the QAQQINI options file (can affect one job or all jobs depending on how it is used).
- Use the SET CURRENT DEGREE SQL statement (V5R4)

As it turns out, SQE and CQE have implemented parallelism differently and this may impact how you choose to use these settings.

The CQE engine implemented parallel queries mostly below the machine interface in the system licensed internal code (SLIC). The database engine used low-level tasks to do work on more than one CPU. These tasks ran in the machine pool rather than the storage pool of the job.

The SQE engine implements parallelism using threads instead of tasks. This provides better work management accounting and control and also ensures the work is done in the same pool as the parent job. The disadvantage of this approach is that each thread consumes an activity level and may impact throughput if the activity level is not configured high enough.

Another difference is the way each of the optimizers handles the *NBRTASKS parameter on the system value, CHGQRYA, and in the QAQQINI file. Here is an example using CHGQRYA that tells the query optimizer to use ten background tasks or threads to run the query:

CHGQRYA DEGREE(*NBRTASKS 10)

The CQE optimizer used the DEGREE value as **advice** and would use **at most** ten tasks. SQE on the other hand, **trusts** that the person using the *NBRTASKS value knows what they are doing and will honor the request to **create exactly** ten threads to implement the query. This may result in some unexpected performance of SQL statements if a query that was previously run by the CQE is now processed by the SQE. The performance in some cases could be better, and in others, worse.

Here are some general guidelines for using the SMP feature:

- If you choose to implement parallelism globally via the system value or a QAQQINI file in the QUSRSYS library, use *OPTIMIZE.
- Unless you have done extensive experimentation to prove that a particular number of tasks or threads works well, let the optimizer choose the level of parallelism by *not* using *NBRTASKS.
- If your SQL statements are using CQE and parallelism, make sure your machine pool is large enough to handle the extra activity.
- If your SQL statements are using SQE and parallelism, make sure your activity levels are set high enough to allow for the extra threads to run without interfering with other work.

Allowing DB2 to use multiple processors is one of the easiest ways to speed up SQL queries. Given there are two SQL query engines however, you do need to be aware of the differences if you use advanced parallel options. The good news is there are plenty of options and you can control when and who uses the processing power on your system.

Good luck using this powerful feature.

Mark

CTO MEMO

Keep your eyes open for announcements from Centerfield Technology in the next couple of months! We have some exciting developments that I think you will be as thrilled about as I am. First and foremost, you'll hear the details about our upcoming release of HomeRun 6.0.

Here at Centerfield we are spending a lot of time making the complex and tedious job of tuning SQL and DB2 for i5/OS much easier. It's all about simplicity and saving your time because you tell us that it is impossible to be an expert in all the software and hardware you work with every day. We are focused on providing value by driving down the effort, expertise, and risk of managing the databases that your business-critical applications depend upon. To accomplish those goals we are working to deliver tools that:

- ✦ Provide answers and advice – not simply providing tools that show you screens and screens of raw data or pretty pictures.
- ✦ Automating tasks – not simply providing tools that require you to point and click to get anything done.
- ✦ Allow you to implement advice with a minimal amount of effort and risk – rather than leaving you to figure out how to do it on your own.
- ✦ Work well in high volume environments – not tools that drag your system down every time you use them.

Best Regards,

Mark L. Holm
Chief Technology Officer
Centerfield Technology, Inc.

INSURANCE POLICY

Quick, answer this, how much does one hour of down time of your System i box cost your company?

Is it worth $10K…? $20K...? $30K…?

Establishing cost of downtime enables you to establish value of not having unnecessary outage. I realize unplanned outage in System i world is a rarity, but System i can go down due to critical storage condition (running out of DASD). It will warn you at some pre-configured threshold (90%-95% are most common settings), but what if you need or want to know of a DASD spike long before that?

While there are number of home grown and vendor tools for **strategic** disk growth monitoring (i.e. when do I need to purchase more DASD due to normal data growth), there really are no **tactical** tools out there to figure out quirky and unique DASD spikes.

Here comes disk/HUNTER.

It sits quietly in the background until a threshold trigger you've configured fires. It then collects I/O metrics of the deepest possible sort using Performance Explorer (PEX) trace for about 5 minutes, emailing you a report providing exact detail of who, when, what job and what object(s) caused the DASD spike. Details will even list internal system objects that don't map to any external object. These types of spikes can be caused by issues deep inside operating system code, enabling you to provide helpful information to IBM support experts for analysis. These types of internal system object leaks can't be audited by anything other than PEX level collection.

More commonly however, issues are caused by applications going haywire, bad programs, poorly tuned SQL queries, issues with the Java garbage collector, unmonitored IFS growth etc.

Just the other day we got a call from a prospect that we talked about disk/HUNTER but opted not to follow through with it due to budget constraints. When he called, he didn't ask for an evaluation or anything. Instead he said just send me the CD and bill me. When we pushed him to elaborate a bit on sudden shift in urgency, he said a single job running SQL ate up 1TB of his storage. Obviously, this issue is less likely to repeat now that he's current on PTFs and has built some appropriate SQL indexes, but if he had disk/HUNTER in place he would have known about the issue way before it ate up 1TB of his system and slowed down operations to a crawl.

If you're the type of person that likes to keep on top of things and knows a value of good insurance policy, put disk/HUNTER on your machine. The cost to benefit ratio really makes it a no brainer.

Delete Old IFS Files

This article will address a request from a reader to write about a method to find old objects in the IFS for cleanup purposes.

What is Qshell and do I have it installed?

http://www-1.ibm.com/support/docview.wss?uid=nas16e6a58b238b2c3788625722d00544d5c&rs=110

After glancing at the above listed link, I can see iSeries folks already rolling their eyes, so suffice it to say that you can use Qshell to manage IFS effectively and relatively easily. You don't even need to become an expert on the Unix command interface (which in my opinion can't hold a candle to the iSeries command interface) to use Qshell for basic needs like deleting old IFS files or listing your largest IFS objects.

The reason I am leading with the Qshell solution is because it's a natural interface to most folder based file architectures like Unix, Linux and other *NIX types. So let's get to the gold winner immediately:

STRQSH CMD('find /home/elvis -atime +180 -print | xargs rm –rf')

NOTE: If run against the root directory (*/*) and depending on the number of files in the IFS, this may be a long running command.

Let's dissect this nugget. Our *find* command takes input from the folder */home/elvis* and searches the folder and all of its subdirectories while taking additional arguments into account. In our case additional arguments are –*atime* switch which directs the find command to consider the last time files were used or accessed versus *–mtime* (data changed) or *–ctime* (status changed). The next argument *+180* tells the *find* command to look at files older than 180 days.

Three arguments: *–print | xargs* work in conjunction to pipe (*|*) output (*-print*) of the *find* command as a single string (*xargs*) argument to the next command of interest: remove file (*rm*). The remove command is then instructed to remove files recursively in all directories (*r*) without prompting (*f*). The key piece in this command is obviously the pipe feature, allowing us to feed the *rm* command easily. The not-so-obvious benefit of using *xargs* is that on the iSeries this will result in a single job to remove all of the found files, versus a job per file. Note that the *xargs* area has a size limit and although it's quite high on the iSeries, it is possible to hit it if a lot of files meet your search criteria.

There is an alternative script that would feed one file at a time to the *rm* command and remedy the potential *xargs* limitation but it would result in a job-per-file scenario. If there are alot of files that satisfy the criteria, this solution would be a terrible performer on the iSeries due to the high job startup cost versus the lightweight Unix processes. Since we're all about performance, I'm not going to illustrate that method.

If I look back at the reader's original request, he only asked for a way to get at old files, not necessarily automatically delete them. To achieve that, I'm going to slightly modify the original command:

CRTPF NoQtempLib/YourFile RCDLEN(5000)
STRQSH CMD('find /home/elvis -atime +180 -print > /QSYS.LIB/NoQtempLib.LIB/YourFile.FILE/YourFile.MBR')

Make the temporary file (*YourFile*) length as long as your longest fully qualified IFS file path. Now you have the output of old files which you can process in CL (*RMVLNK*) or simply report on to your users for discretionary deletion.

I don't have Qshell installed, now what?

Fear not, there are always alternatives on our trusty iSeries.

With V5R3 IBM has included RTVDIRINF and PRTDIRINF with should look very familiar to folks used to RTVDSKINF and PRTDSKINF. This will provide all the detail you want.

If you're up for a little bit of programming, any of the Unix apis like readdir(), opendir(), stat() etc. are available in all HLL programming languages (RPG,C,COBOL etc.). With access to these, you can easily create your own commands, especially considering the wealth of examples the Unix aficionados have placed on the World Wide Web. Alternatively, there are unique iSeries APIs that perform well in relation to other approaches: Qp0lProcessSubtree, Qp0lGetAttr, Qp0lUnlink.

I was going to write a command but then realized that the well known iSeries expert, Scott Klement, has already done so:

http://www.systeminetwork.com/artarchive/newsletter/w/1001/a/52993/index.html

Other non-Qshell nuggets

- Recursively delete directories
  DSPF /
  Once you locate the directory you want to delete, take option ***9***

- Get IFS file size metrics
  call qsrsrv parm("METRICS" '/mydir/mysubdir')
  then do *DSPJOB* and take option *4* to see a *QSRSRV* spooled file with sizes etc.

- IBM's IFS toolset
  Download the toolset by going here:
  ftp://testcase.boulder.ibm.com/as400/fromibm/ApiSamples/
  Click on the IFSTOOLS.SAVF file.

  Check its documentation here:
  http://www-912.ibm.com/s_dir/slkbase.nsf/0/3976fed8ab10134b862568b60071ccd3?OpenDocument

Elvis

(SQL in CL...What a Variety! — Continued from page 1)

number of versions of this technique floating on the Internet. I haven't taken the time to look at all of them.

Qshell

What is Qshell and do I have it installed:

http://www-1.ibm.com/support/docview.wss?uid=nas16e6a58b238b2c3788625722d00544d5c&rs=110

Qshell is a command interpreter for folks used to UNIX, but it does run natively on the System i (i.e. it's not an environment for running AIX binaries like PASE is).
If you have Qshell installed, it's very straightforward to use, as illustrated in the following example:

*STRQSH CMD('db2 "CREATE TABLE RUNSQL.T8 AS (SELECT * FROM QSYS2.QSQPTABL) WITH DATA"')*

Obviously, you could create an input parameter on the fly in your CL programs.
When I ran this interactively I got the following display:

```
  **** CLI ERROR *****
         SQLSTATE: 01567
NATIVE ERROR CODE: 7905
Table T8 in RUNSQL created but was not journaled.
Press ENTER to end terminal session.
```

The textual message clearly indicates that a table was created, but since the RUNSQL library is not a true SQL collection we get the warning message that automatic journaling could not be put into effect. I verified that SQLSTATE means the same thing by checking it on the IBM's DB2 SQL message finder:

http://publib.boulder.ibm.com/iseries/v5r2/ic2924/index.htm?info/rzala/rzalafinder.htm

Remember that I said Qshell runs natively on the System i? What actually happens is that Qshell spawns a separate process (submits another job) that calls an OS program object to execute the request. That program object is QSYS/QZDFMDB2. So even if you don't have Qshell installed you may try your luck by calling it directly:

*CALL QZDFMDB2 PARM('CREATE TABLE RUNSQL.T9 AS (SELECT * FROM QSYS2.QSQPTABL) WITH DATA')*

The drawbacks of Qshell? It's not always installed on customers' boxes and since it's an interpreted language environment, it is a bit slow to get going. My pet peeve with it is error handling. Qshell just doesn't lend itself to the powerful and easy error handling interface we're used to in CL (i.e. MONMSG).
`

EXECUTE IMMEDIATE

You may be wondering why I bring the EXECUTE IMMEDIATE SQL command into a CL related article, being that its use is reserved for HLL programs with embedded SQL. Quite simply, I mention it because you can execute an arbitrary non-SELECT statement using EXECUTE IMMEDIATE with very little code while at the same time accessing very powerful error handling and additional feedback.

When checking the SQLCA's SQLCODE variable, the command processing program can throw an exception, return a RETURN variable, or just simply proceed with the execution. You can extend it further to return number of rows updated by checking the SQLDA data structure among other things.

Rather than reinvent the wheel, let me point you to Michael Sansoterra's article which illustrates the power and ease of EXECUTE IMMEDIATE by wrapping it into a very CL friendly RUNSQL command:

http://www.mcpressonline.com/mc/.6ae7d3dd

It's hard to beat this particular approach. About the only drawback I see to this interface is that SELECTs are not allowed and the file output option would be tricky to implement, although still possible with a CREATE TABLE AS … or CREATE VIEW AS … type statement.

(Continued on page 7)

Centerfield TECHNOLOGY

strategic tools for teams strategic applications

(SQL in CL...What a Variety! — Continued from page 6)

CLI

That leads us to the final frontier, CLI (Call Level Interface). Let me start by saying that in no way am I advocating this method if you have access to ***any*** HLL programming language. HLLs are much more versatile, more powerful and easier to use when running embedded SQL statements. That said, if you must do it in CL and want the full power of embedded SQL, CLI is probably the one and only way to do it.

I like to describe CLI as a native ODBC interface. Before V5R3, this particular set of APIs was not available in CL since a large number of functions required passing parameters by value, and CL only allowed passing parameters by reference. With V5R3 it is now possible to pass parameters by value in CL (kudos to IBM!).

With CLI you can literally do anything that's valid in SQL! You can even call stored procedures and handle returned result sets in a nice fashion. The possibilities are endless, including writing a generic SQL command executor.

I took a peek at IBM's website to see if they have any examples you could look at. I found one, but it was written in C:

http://publib.boulder.ibm.com/infocenter/iseries/v5r3/topic/cli/rzadpexembeddedsql.htm#rzadpexembeddedsql

I couldn't find any in CL, so I figured I'd just rewrite what they have in CL. Here is the end result that, while definitely in need of some tweaking, still amply illustrates the power of a CL/CLI combo. To compile it use:

CRTBNDCL PGM(qgpl/CLI) SRCFILE(yourLib/QCLSRC)

NOTE: I was using a V5R4 system to create this sample program, but it ought to work on V5R3 as well

The source member type must be CLLE as I'm calling CLI functions as procedures. The resulting program requires no parameters so simply call it as:

CALL qgpl/CLI

```
/* Start of CL source */
PGM
dcl &henv     *int
dcl &hdbc     *int
dcl &hstmt    *int
dcl &server   *char 10 '*LOCAL'
dcl &uid      *char 30
dcl &pwd      *char 30

dcl &id       *int
dcl &name     *char 51
dcl &namelen  *int
dcl &intlen   *int
dcl &scale    *char 2 X'0000' /* SMALLINT */

dcl &sql_nts  *int value(-3)
dcl &cmtAttr  *int value(0)
dcl &noCommit *int value(1)
dcl &retcode  *int
dcl &success  *int value(0)
dcl &scswinfo *int value(1)
```

(Continued on page 8)

(SQL in CL...What a Variety! — Continued from page 7)

```
dcl &sql      *char 100
dcl &one      *int  value(1)
dcl &two      *int  value(2)
dcl &fifty    *int  value(50)
dcl &fiftyone *int  value(51)
dcl &sqlclong *int value(4)
dcl &sqlcchar  *int value(1)
dcl &sqlvchar *int value(12)
dcl &intchar  *char 10
dcl &nullterm *char 1 x'00'

chgvar &server (&server *Tcat &nullterm)

/* allocate an environment handle */
CALLPRC PRC('SQLAllocEnv') PARM((&HENV))
/* allocate a connection handle */
CALLPRC PRC('SQLAllocConnect') PARM((&HENV *BYVAL) (&HDBC))
/* connect to *LOCAL database (meaning this LPAR) */
CALLPRC PRC('SQLConnect') PARM((&HDBC *BYVAL) (&SERVER) (&sql_nts *byval) +
   (*OMIT) (&SQL_NTS *BYVAL) (*OMIT) (&SQL_NTS *BYVAL)) RTNVAL(&RETCODE)
IF (&RETCODE *NE &SUCCESS) THEN(DO)
  SNDPGMMSG 'SQLConnect failed'
  goto cleanup
ENDDO

/* turn off commitment control */
CALLPRC PRC('SQLSetConnectAttr') PARM((&HDBC *BYVAL) (&cmtAttr *byval) +
          (&noCommit) (0))
/* allocate a statement handle */
CALLPRC PRC('SQLAllocStmt') PARM((&HDBC *BYVAL) (&HSTMT))

chgvar &sql 'CREATE TABLE NAMEID (ID integer, NAME varchar(50))'
/* execute the sql statement */
CALLPRC PRC('SQLExecDirect') PARM((&HSTMT *BYVAL) (&sql) (&sql_nts *byval)) +
       RTNVAL(&RETCODE)
IF ((&RETCODE *NE &SUCCESS) *and (&retcode *ne &scswinfo)) THEN(DO)
  SNDPGMMSG 'SQLExecDirect failed'
  goto cleanup
ENDDO

/* illustrate the use of SQLPrepare/SQLExecute method */
chgvar &sql 'INSERT INTO NAMEID VALUES (?, ?)'
/* prepare the insert                              */
CALLPRC PRC('SQLPrepare') PARM((&HSTMT *BYVAL) (&sql) (&sql_nts *byval)) +
       RTNVAL(&RETCODE)
IF (&RETCODE *NE &SUCCESS) THEN(DO)
  SNDPGMMSG 'SQLPrepare failed'
  goto cleanup
ENDDO

/* Set up the first input parameter "id"                   */
chgvar &intlen 4
CALLPRC PRC('SQLSetParam') PARM((&HSTMT *BYVAL) (&ONE *BYVAL) (&sqlclong *byval) +
        (&sqlclong *byval) (&sqlclong *byval) (&scale *byval) (&id) (&intlen))
/* Set up the second input parameter "name"                 */
chgvar &namelen -3
CALLPRC PRC('SQLSetParam') PARM((&HSTMT *BYVAL) (&two *BYVAL) (&sqlcchar *byval) +
        (&sqlvchar *byval) (&fifty *byval) (&scale *byval) (&name) (&namelen))

/* now assign parameter values and execute the insert             */
```

(Continued on page 9)

(SQL in CL...What a Variety! — Continued from page 8)

```
chgvar &id 500
chgvar &name 'Babbage'
/* execute insert statement */
CALLPRC PRC('SQLExecute') PARM((&HSTMT *BYVAL))
RTNVAL(&RETCODE)
IF (&RETCODE *NE &SUCCESS) THEN(DO)
  SNDPGMMSG 'SQLExecute INSERT failed'
  goto cleanup
ENDDO

chgvar &sql 'SELECT ID, NAME FROM NAMEID'
/* execute SELECT statement */
CALLPRC PRC('SQLExecDirect') PARM((&HSTMT *BYVAL)
(&SQL) (&SQL_NTS *BYVAL)) +
    RTNVAL(&RetCODE)
IF (&RETCODE *NE &SUCCESS) THEN(DO)
  SNDPGMMSG 'SQLExecDirect SELECT failed'
  goto cleanup
ENDDO

/* Binding first column to output variable "id"            */
CALLPRC PRC('SQLBindCol') PARM((&HSTMT *BYVAL)
(&one *byval) (&SQLclong *BYVAL) +
    (&id) (&sqlclong *byval) (&intlen))
    /* Binding second column to output variable "name"
*/
CALLPRC PRC('SQLBindCol') PARM((&HSTMT *BYVAL)
(&two *byval) (&SQLcchar *BYVAL) +
    (&name) (&fiftyone *byval) (&namelen))

CALLPRC PRC('SQLFetch') PARM((&hstmt *byval))
chgvar &intchar &id
SNDPGMMSG MSG('Result of Select: id = ' *cat &intchar
*cat ' name = ' +
               *cat &name)

CLEANUP:

/* free statement handle */
CALLPRC PRC('SQLFreeStmt') PARM((&HSTMT *BYVAL)
(&ONE *BYVAL))
/* disconnect from the database */
CALLPRC PRC('SQLDisconnect') PARM((&HDBC *BYVAL))
/* free the connection handle */
CALLPRC PRC('SQLFreeConnect') PARM((&HDBC
*BYVAL))
/* free the environment handle */
CALLPRC PRC('SQLFreeEnv') PARM((&Henv *BYVAL))

ENDPGM
/* End of CL source */
```

Here is a list of available CLI functions:

http://publib.boulder.ibm.com/infocenter/iseries/v5r4/topic/cli/rzadphdapi.htm

To look up values of the constants that these functions can accept as arguments (i.e. SQL_NTS stands for Null Terminated String) I went to the QSYSINC/H/SQLCLI header file.

I didn't go as far as creating a generic RUNSQL command leveraging CLI, but it is definitely doable and I leave it as a reader exercise. I find the use of CLI more valuable when you need to consume a stored procedure result set output and don't have access to HLL for the task.
While it's undeniably the most powerful CL/SQL combo, CLI is also the most complex of the methods I mentioned. That and CL's inability to use it prior to V5R3 are its biggest detractors.

Summary

I illustrated a number of methods to execute SQL in CL. As always, I urge you to use the best tool for the task at hand! HLLs have much better support for embedded SQL so that's what I primarily use. Every once in a while though, I need to run SQL statements in CL and always turn to one of the methods I've outlined above.

My sincere hope is that I've opened some eyes out there and eased SQL adoption within the System i Community.

Happy reading.

Elvis Budimlic
Development Director