SQL IQUERY SCRIPT

The Modern SQL Language for IBM i

First Ed. Jan 2023

THINKING IN SQL

As IBM i developers we often tend to think of solutions by considering how we can solve a problem using RPG IV or CL. With SQL iQuery and even with RPG IV and embedded SQL we developers need to starting *Thinking in SQL*™

This means considering a data-centric approach to problem solving. Rather than treat the data as a repository that needs to be read and then processed, think of how to extract the results you need for your endgame. That is, *let the database query engine do the work for you*.

With few exceptions, legacy applications that only process data can be refactored/modernized so that they only use SQL and specifically, SQL iQuery Script. In addition, generating classic Reports for end-users can also be produced using SQL iQuery Script. In fact, iQuery Script is the only way I create reports for an end-user today. I haven't written an RPG print program for report purposes in 10 years.

Scenario 1:
The sales executive needs a report showing sales by region and summarized by sales representative within that region. Of course, they want an interactive (classic Inquiry) result as well as a print option to produce the report.

You start building an RPG IV with DDS for the Display file application. After a couple weeks you deliver the application to the end-user. They use it for a week or two and then state, "What I really want is this information in Excel, can you do that?"

Scenario 2:
The Purchasing department would like a consolidated report containing the sales for certain items across all distribution centers. This requires that you pull in data from remote IBM i partitions. So, you spend several weeks building CL programs to prompt them for the items and pull in the data from each remote partition/server, create RPG or perhaps OPNQRYF consolidation routines, and build a cool looking DDS-based PRTF Report for them.

After delivering this awesome solution, the ask if they can get it delivered via EMAIL and in Excel format.

Scenario 3: Thinking in SQL

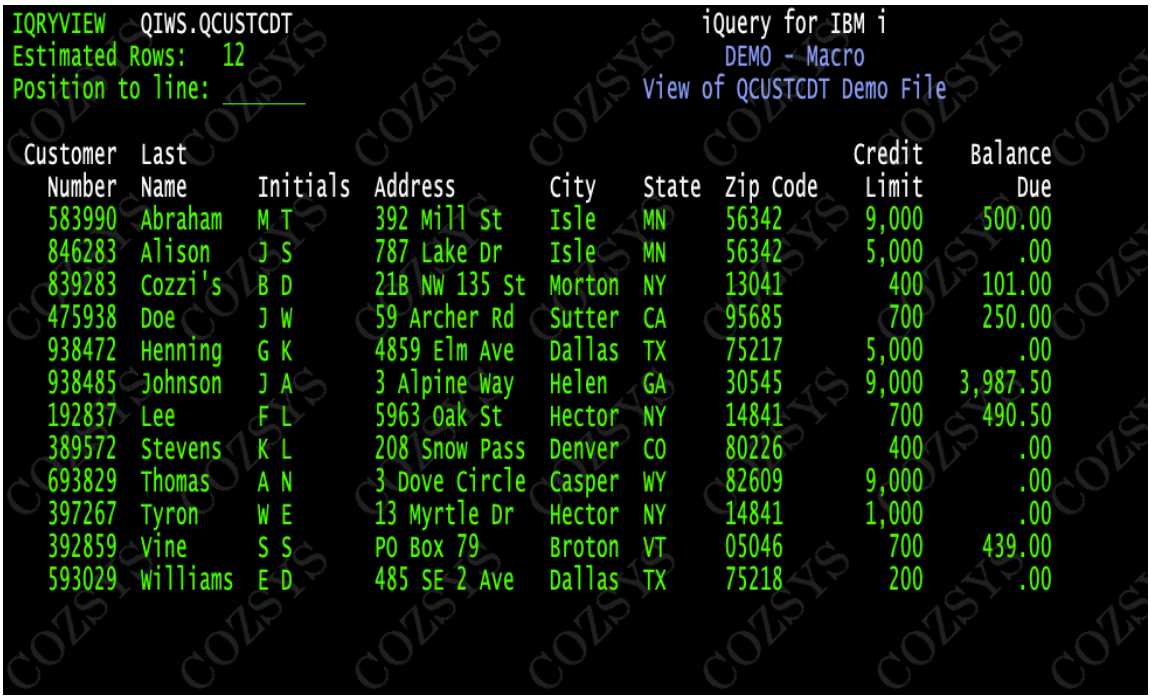
When an end-user has a request for an inquiry program or report, you should anticipate that the end-product requirements will evolve after they use it, or in some cases before you deliver it. If you start the design phase by thinking in SQL you end up with a flexible outcome that can easily adapt fluid situation.

For example, in scenario 1, above, if you created the report using SQL iQuery script, when the end-user changed the result to Excel, you could have said "Yes" and then simply added OUTPUT(*EXCEL) to the RUNiQRY command or embedded it into the iQuery Script as the default output media using #DFTOUTPUT EXCEL.

SQL IQUERY SCRIPT USERS GUIDE

This document describes the SQL iQuery Script syntax and development process. It does so from the perspective of an IBM i RPG IV programmer. The reader does not need to be an advanced RPG IV developer, but references to RPG IV are used for context through this document.

```
RUNiQRY 'select * from qiws.qcustcdt'
```



IQRYVIEW QIWS.QCUSTCDT — iQuery for IBM i
Estimated Rows: 12 — DEMO - Macro
Position to line: _____ — View of QCUSTCDT Demo File

Customer Number	Last Name	Initials	Address	City	State	Zip Code	Credit Limit	Balance Due
583990	Abraham	M T	392 Mill St	Isle	MN	56342	9,000	500.00
846283	Alison	J S	787 Lake Dr	Isle	MN	56342	5,000	.00
839283	Cozzi's	B D	21B NW 135 St	Morton	NY	13041	400	101.00
475938	Doe	J W	59 Archer Rd	Sutter	CA	95685	700	250.00
938472	Henning	G K	4859 Elm Ave	Dallas	TX	75217	5,000	.00
938485	Johnson	J A	3 Alpine Way	Helen	GA	30545	9,000	3,987.50
192837	Lee	F L	5963 Oak St	Hector	NY	14841	700	490.50
389572	Stevens	K L	208 Snow Pass	Denver	CO	80226	400	.00
693829	Thomas	A N	3 Dove Circle	Casper	WY	82609	9,000	.00
397267	Tyron	W E	13 Myrtle Dr	Hector	NY	14841	1,000	.00
392859	Vine	S S	PO Box 79	Broton	VT	05046	700	439.00
593029	Williams	E D	485 SE 2 Ave	Dallas	TX	75218	200	.00

Resources

SQL iQuery is available from www.SQLiQuery.com or it may be used at no charge on the PUB400.com portal on a current IBM Power server running the latest version of IBM i.

Uses

You can process SQL using SQL iQuery through any of the following methods:

- CL Command Entry
- CL Programs
- Source Members ("Scripts")
- Web (CGI interface)
- HLL using our proprietary APIs (rare)

In this book, I will focus on the first three uses and specifically the use of Source Members to create SQL iQuery Scripts. But let's review the first two (Command Entry and CL programs) before continuing.

Running SQL from Command Entry

SQL iQuery allows users to run SQL statements from the CL Command Entry screen. To do this the RUNiQRY (Run SQL using iQuery) CL command is provided. This command has all the options needed to process any SQL statement from command entry.

```
RUNiQRY 'select * from qiws.qcustcdt'
```

In the above example, the SQL SELECT statement is used to query the QCUSTCDT file stored in the QIWS library. This happens to be a demo file shipped with the operating system, so querying it is probably allowed on your system. Note that SQL iQuery is not intended to be used to "compile" SQL/PL source to create SQL Procedures or Functions.

The output from the above SELECT statement is, by default, routed to the screen. Most users today utilize IBM ACS for 5250 emulation but there are other emulators out there, such as the MochaSoft TN5250 which is widely used.

IBM ACS users typically have both *DS3 and *DS4 modes available to their 5250 sessions. For that reason, SQL iQuery checks the screen capabilities and if *DS4 is supported, the output will use the full 132 characters and 27 rows of the screen.

The advantage of using SQL iQuery over the native RUNSQLSTM CL command is that RUNiQRY supports the SELECT and WITH statements, it can process SQL scripts dynamically, and it can direct the output of a SELECT to a several IFS file formats in addition to the standard display and print formats.

For example, using the OUTPUT parameter of the RUNiQRY CL command, you can direct the output to a variety of media format such as print, a data area, Excel, PDF, CSV, JSON, raw text, and more.

Running SQL within CL Programs

SQL iQuery allows users to run SQL statements within CL programs. Again, the RUNiQRY CL command is used to provide the capability. The advantage of using SQL iQuery is that the CL program can "see" the SQL statement that's being run. Alternatively, you can store SQL statements in source file members and run them using SQL iQuery:

```
RUNiQRY SQL('UPDATE QIWS.QCUSTCDT SET CDTLMT = 0 WHERE BALDUE > 0')
```

or

```
RUNiQRY SRCFILE(prodsrc/qsqlsrc) SRCMBR(UPDCSTBAL)
```

With SQL iQuery, CL programmers can embed the SQL statement directly into the CL program. They can dynamically build the statement with embedded CL variables just like any other CL *CAT operation or specify the entire statement as a literal.

There are advantages of using a Source File member to store the SQL statements you want to run in CL or elsewhere. Source file members can contain simple SQL statements or simple to complex SQL iQuery Scripts. For general single-statement use, however, RUNiQRY gives you the advantage of being able to specify the SQL statement directly.

> **Tip**: You can redirect the output of a single-record SELECT statement to a data area using SQL iQuery's OUTPUT(*DTAARA) parameter and specifying the data area name on the OUTFILE parameter. This allows you to then pull in that information using the RTVDTAARA CL command.

SQL Source File Members

As mentioned, SQL statements may be stored in standard source file members and run using the RUNiQRY CL command. We call those source members *SQL iQuery Scripts*. The SQL statements in a source member are run in sequence by the SQL iQuery Script processor. The final statement in the script however, is passed back to the SQL iQuery engine for processing (i.e., it is returned to the RUNiQRY command for processing).

For example, suppose you have need to run an SQL DELETE or perhaps an UPDATE statement and then you want a table to be queried with a SELECT statement and the results sent to an Excel-compatible file on the IFS. You would need an SQL iQuery Script (i.e., source file member) to store and then run those statements.

```
01)DROP TABLE QTEMP.PGMREFS;
02)CL: DSPPGMREF PRODLIB/ORD* OUTPUT(QTEMP/PGMREFS);
03)SELECT WHPNAM,WHFNAM,WHLNAM,WHOTYP,WHFUSG
04)    FROM QTEMP.PGMREFS;
```

The DROP statement and the CL: directive (lines 1 and 2) are processed by SQL iQuery script. The final statement (lines 3 and 4) is passed back to and is processed by the RUNiQRY command as if it were coded as a literal on the RUNiQRY command itself:

```
RUNiQRY SQL( <the last statement> )
```

This last statement, therefore, benefits from the various options specified on the RUNiQRY CL command, such as OUTPUT, EMAIL, Titles, etc.

The only exception to this last statement rule is with nested scripts. In that context, all statements in a nested script source member are run by the SQL iQuery Script processor. More on this later when we cover nested scripts.

SQL iQuery Scripts are a great way to set up the data, output media, email recipients, and other attributes. Here is a simple example.

Source Mbr: MYSTUFF/QSQLSRC(DEMO)

```
01)#COLTOTALS 8, 9
02)#NUMEDIT 8, 9
03)#DFTOUTPUT *EXCEL
04)SELECT CUSNUM    as "Customer            Number",
05)       LSTNAM    as "Last                Name",
06)       INIT      as "Initials",
07)       STREET    as "Address",
08)       CITY      as "City",
09)       STATE     as "State",
10)       DIGITS(ZIPCOD) as "Zip Code",
11)       CDTLMT    as "Credit              Limit",
12)       BALDUE    as "Balance             Due"
13) from qiws.qcustcdt
14) order by lstnam;
```

The final statement is the SELECT statement (line 3). When that SELECT is run, it is passed back to the SQL iQuery processor and is run as if it were passed directly to the RUNiQRY command itself.

Lines 1, 2, and 3 are SQL iQuery Script directives. They cause attributes to be set and the output media to be selected.

Line 1 contains the #COLTOTALS directive. It identifies one or more columns that should be accumulated. This applies to output that is printed or sent to MS Excel. Either the relative column number (as shown in this example) or the actual column name may be specified. Multiple column IDs are separated by a comma. Today, SQL for IBM i supports OLAP functions which were not yet available when SQL iQuery was created, so we built it into iQuery Script.

Line 2 contains the #NUMEDIT directive. It identifies one or more columns that should have basic numeric editing applied to them. By default, iQuery avoids thousands notation (that is the comma for most countries, and period for others). This directive applies a basic numeric edit to the data before it is written to the output media. For example, normally 1234.50 would be written, but when #NUMEDIT identifies that column, the value is written as 1,234.50 instead.

Line 3 contains the #DFTOUTPUT directive. It identifies the output media to be used by default. This means that if the RUNiQRY command is specified with no output parameter, or with its default value of OUTPUT(*DFT), then this directive will assign evoke OUTPUT(*EXCEL) as the output media. If however, the OUTPUT parameter has some other value, then #DFTOUTPUT is ignored. Note: There is also an #OUTPUT directive that forces the output media to whatever is specified on that directive, ignoring the OUTPUT parameter entirely.

Source Members

Source members are used for SQL iQuery Scripts which can contain any number or statements or "lines of code". Source members can be any length (width). SQL iQuery Scripts are not limited to 79 columns. You can create a source file for use as an SQL iQuery Script with any supported length. Note that SEU has a hard limit of 240 bytes for source file record length.

When creating source files, I tend to create them with a 112-byte record length:

```
CRTSRCPF FILE(MYSRC/QSQLSRC) RCDLEN(112)
```

This gives you 100 bytes for the SQL code and accommodates the 12 bytes used by line change date, line sequence number fields. As a general habit, I tend use 112 or simply max out at the 240-byte SEU limit instead of the legacy 92-byte CRTSRCPF command default.

SQL iQuery Script source members can be edited with any of the tools available today such as SEU, RDi, and Microsoft Visual Studio CODE using the *CODE for i* plug-in. Unlike SEU, the other editors do not have a line-length limitation. Note that if you set the SEU Type (source type) to SQL, then RDi and VS CODE will render the code using syntax highlighting when editing SQL iQuery Scripts.

Column Headings

A quick note on column headings of your output. Db2 for i SQL uses the IBM i DDS Column Headings. The COLHDG keyword in DDS is used to declare the column headings used by various tools (such as Query/400) and SQL iQuery. Using the SQL *LABEL ON COLUMN* statement you can also set the same column headings text. In addition, when a SELECT statement is run, if you include a correlational name via the "AS" clause for a column, that correlational name is used as column heading.

Each of the 3-lines of column headings may be specified on the AS clause by spacing then out in 20-byte strings. For example, the Customer Number (CUSNUM) heading could be specified as:

```
LABEL ON COLUMN QIWS.QCUSTCDT (CUSNUM as 'Customer            Number');
```

In the IBM i DDS source code, this would be specified as:

```
A            CUSNUM         6S 0       COLHDG('Customer' 'Number')
```

Each of the 3 column heading lines occupies 20 bytes in the database object. Only 2 of the 3 headings are illustrated above. To specify headings using a runtime SQL SELECT statement, you specify a single string of up to 60 bytes, with each 20-bytes containing one of the three column heading components. However, unlike LABEL ON and the COLHDG keyword, in this context double-quotes are used instead of apostrophes.

```
01)  --                  *........1.........2.........3.........4
02) SELECT CUSNUM    as "Customer            Number",
03)        LSTNAM    as "Last                Name",
04) ... etc.
```

SQL iQuery inserts the column headings into the result set for SELECT statements where possible. They are used for output to the screen/display, print, Excel, CSV and even JSON. Other SQL tools, such as the legacy STRSQL and IBM ACS RUNSQL Scripts (GUI) also use this information for column headings.

SQL iQuery Scripting

The demo source member illustrated in the previous section is a good example of a simple SQL iQuery Script.

SQL iQuery script has a full set of conditional logic controls as well as being an SQL processor. That is, it has IF/ELSE/DO control statements that work similar to their corresponding RPG IV opcodes. In fact, iQuery Script logic controls are a sort of hybrid of RPG IV and SQL. That is in addition to traditional condition testing, SQL User Defined Functions (UDF) as well as entire SELECT statements may be included. For example, you can code the following:

```
01) If EXISTS (Select * from QIWS.QCUSTCDT WHERE BALDUE > 0);
02)  Foreach select cusnum, lstnam, state, baldue
03)              INTO :custNo, :lastName, :State, :BalDue
04)           FROM QIWS.QCUSTCDT
05)           WHERE BALDUE > 0;
06)    If (&STATE = 'NY');
07)      #MSG Balance Due found for Customer &CUSTNO in New York
08)    elseif (&STATE = 'CA');
09)      #MSG Balance Due found for customer &CUSTNO in California
10)    else;
11)     #MSG Balance Due customer &CUSTNO found in &STATE
12)    endif;
13)   endFor;
14) Endif;
```

In this example, line 1 checks if any rows exist in the QCUSTCDT file that have a balance due. If so, it proceeds to the next statement (line 2) which has the FOREACH command. Otherwise, it jumps to the ENDIF statement on line 14.

The FOREACH opcode processes each row of the SQL SELECT statement using a *cursor*. It's all transparent to the programmer, making it easier to code.

Each row that's fetched is stored into the host variables of the INTO clause, just like RPG IV. You can use the standard SQL host variable prefix of the colon (as illustrated here) or the SQL iQuery Script variable prefix of the ampersand (similar to CL variables). Both work in this context. Outside of the INTO clause, the ampersand (&) symbol is required to identify variables. See *Session Variables* later in this section.

The #MSG on lines 7, 9 and 11 is an SQL iQuery Script directive. It is used to write text directly to the joblog of the job running the script. If you were to run this script from the Command Entry display, the output would look similar to the following:

```
iQScript: COZTEST/QSQLSRC(CUSTBALDUE)
```

```
Balance Due found for Customer 839283 in New York
Balance Due customer 392859 found in VT
Balance Due customer 938485 found in GA
Balance Due found for customer 475938 in California
Balance Due found for Customer 192837 in New York
Balance Due customer 583990 found in MN
```

The first line is an iQuery Script-generated message that indicates the name of the source file and member being processed. This is logged whenever a SQL iQuery Script source member is loaded. It comes in handy when debugging scripts or checking joblogs.

The output shown above is the result of the #MSG directives. Everything to the right of the #MSG directive itself is written to the joblog as an INFO message. Note that #MSG, #SNDMSG and #JOBLOG are synonyms and may be used based on your preference.

SQL iQuery Script ignores upper/lower case that is not quoted. The names of Script commands, directives, opcodes, logic conditional statements along with variable names may be specified in upper/lower case, which is ignored.

Session Variables

SQL iQuery Script supports fields or *variables* to store data similar to other languages. These variables are formally named *Session Variables.* Session Variable names may be up to 30 characters in length, and must start with the Ampersand symbol (&) and be followed at least one alphabetic characters. That is &A is valid, but &8 is invalid. Subsequent characters in the name may consist of A-Z, 0-1, and the underscore (_) symbol. The first character must be A-Z, the last character may not be an underscore.

The Session Variable naming rules are:

- They must begin with an Ampersand followed by at least 1 letter (A-Z).
- The rest of the name may be A-Z, 0-9 or the underscore.
- Upper/lower case is ignored.
- The final character may not be an underscore.
    - VALID: &HELLO_WORLD
    - INVALID: &HELLOWORD_

Session Variable are implicitly defined "on the fly" similar to how the JavaScript language declares variable. That is, you may explicitly define a session variable using the #DEFINE directive, as follows:

```
#define &varName 'value';
#define &var2 = 'Hello World'
#define &var3 = 12.59
```

The #define command declares the variable and assigns the value to it. If the variable already exists, its value is replaced with the value specified on the #define command. The variable name may be followed by its assigned value. The equals-sign is optional on the #define directive. The semicolon terminator is also optional. The following are all equivalent and valid #define directives:

```
#define &Item_No 12345
#define &ITEM_NO = 12345
#define &ITEM_No = '12345';
#define &item_no '12345';
```

There is another directive that is similar to #define; the #default directive defines a Session Variable and assigns a value to that Session Variable if the variable does not exist. If the session variable exists, the #default statement is ignored.

```
#define &CustNo = 5250;
#msg CustNo = &CUSTNO
#default &CUSTNO = 3741;
#msg CustNo = &CUSTNO
```

The results of the above script would produce the following messages in the joblog:

```
CustNo = 5250
CustNo = 5250
```

The #default directive checked to see if &CUSTNO existed. Since the #define was use to previously declare it, it exists, so the #default directive avoid assigning the value 3741 to &CUSTNO. The output messages show the &CUSTNO session variable is unchanged but the #default directive.

The EVAL opcode may also be used to declare and assign a value to a Session Variable. If the value doesn't exist, it is automatically created.

```
EVAL &CUSTNO = 12345;
```

The above statement assigns the value 12345 to the &CUSTNO session variable. Beginning with SQL iQuery version 7, the EVAL opcode name is optional. The following is also a valid assignment statement:

```
&CUSTNO = 12345;
```

Note: unlike the #declare, #define, #default directives, the EVAL opcode requires an equals sign and terminating semicolon. This is because EVAL supports multiple lines of code for a single statement. The semicolon is used to terminate those statements similar to how it is done in RPG IV free format.

There are two additional methods to define Session Variables and assign a value to them:

1. Specify the SETVAR parameter on the RUNiQRY command.
2. Target the Session Variable using the INTO clause of a SELECT or VALUES statement.

The SETVAR parameter is often used with the #default command. If a Session Variable name is not specified on the SETVAR parameter of RUNiQRY, the #default directive is used to assign a value to that Session Variable. If it is passed on the SETVAR, then #default is ignored.

Session Variables may contain numeric or character data. They may also contain the contents of an IFS stream file. For example, you can read an entire IFS stream file into a single Session Variable using the GETFILE built-in function.

```
eval &notes = getFile('/home/corp/daily_msg.txt');
```

This loads the DAILY_MSG.TXT file's contents into the &NOTES session variable. GETFILE is very useful when using iQuery email capabilities or when using iQuery for Web. It not only reads the file, but it also merges any embedded iQuery Session Variable names. That is, it converts embedded Session Variable names to their content during the load process.

While there is no practical limit to the size of the data stored in a Session Variable, internally each variable's vector memory management is limited to just under 2 gigabytes.

Predefined Session Variables

The following predefined Session Variables are automatically created and assigned a value after each SQL statement is run within an SQL iQuery Script.

&SQLSTATE - Automatically set to the last SQL statement's SQL State. You may use this variable wherever you would normally use the SQL State host variable in RPG or other high-level languages.

&SQLCODE – Automatically set to the last SQL statement's SQL Code. You may use this variable wherever you would normally use the SQL State host variable in RPG or other high-level languages.

&SQLMSG - Automatically set to the last SQL statement's SQL message text (if any). Many SQL STATE codes do not have associated message text, so this Session Variable is often empty.

Undefining Session Variables

To destroy an existing Session Variable, the #undef or #undefine directive may be used. This directive immediately removes the Session Variable from the internal storage collection, destroys and frees its content.

```
#undef &VARNAME
```

In the above statement, the &VARNAME Session Variable is deleted from SQL iQuery Script storage. That session variable name may be re-used for other purposes.

Interpretive Language

SQL iQuery Script is an interpreted language. Each line of code is read, parsed, and processed dynamically at runtime, including all SQL statements. You may insert Session Variables into the script where on any statement. The variable name is replaced with its content at runtime.

When used with directives or iQuery Script commands or conditional logic, you use the session variable name just like any other programming language. When used on an SQL statement, it is wise consider whether to quote or not quote the session variable name itself, based on the context where it is used. In conditional statements and directives quoting a session variable name is almost never required.

When a session variable is used within an SQL statement consider whether the expansion of that variable into its content would cause a syntax error if it were not quoted. Consider the following:

```
01)#define &FOOD = 'SANDWICH'
02)#define &ORDER = 0;
03)#default &REGID = 'CHICAGO'
04)
05)Select ordid, product
06)       INTO &ORDER, &FOOD
07)    FROM &regid.prodData.OnlineOrders
08)    WHERE transID = &TRANSID
09)    LIMIT 1;
10)
11)IF (&FOOD = 'PIZZA');
12)   Update &regid.prodData.orders SET
13)                ITEM = '12',
14)                DESC = '&FOOD'
15)           WHERE ORDID = &ORDER;
16)endIf;
```

Note the use and the quoting of session variables in the above example. One line 6, the INTO clause is actual reading the data into the identified session variables so obviously there use in this context does not call for quoting them. Neither does line 7 and line 12 where the &REGID variable is used to actual compose part of the SQL statement itself (in this case, adding the 3-tier domain/database ID to the qualified name). Line 14 on the other hand, is a character column being assigned the valid of the &FOOD variable. Since DESC = SANDWHICH would cause a syntax error, you have to quote the session variable, as shown.

One exception is when a session variable contains numeric content and is used in an SQL statement with a numeric column. In that situation the session variable name itself should not be quoted. For example.

```
#define &CUSTID = 5250
Select * from prodData.Sales
     WHERE custNo = &CustNo;
```

In the above example, the Session Variable &CUSTNO is expanded to 5250 so the WHERE clause looks like this at runtime:

```
WHERE custNo = 5250;
```

This is valid syntax, so the programmer avoids quoted the &CUSTNO variable in this context.

Using quoted session variable names on conditional statements is supported, but not necessary.

One great example where you wouldn't use the quotes in an SQL statement is when a part of that statement is extracted from a Session Variable. The 3-level database names or 3-tier names is a good example of that. While in SQL/PL and standard SQL on the IBM i platform, the database name must be hard coded into the SQL statement itself, SQL iQuery Script allows programmers to set that part of the statement as a Session Variable and have it resolved at runtime. For example:

```
#define &RMT = 'CHICAGO'
select * from &RMT.qiws.QCUSTCDT;
```

This would expand to:

```
select * from CHICAGO.qiws.QCUSTCDT;
```

In the above example, the session variable &RMT is used to identify the database (IBM i partition) to query. In this case, at runtime the content of &RMT is translated to CHICAGO and added to the SELECT statement shown above.

Now think about extracting the &RMT variable's value from some external source, like a database record, or even the RDBDIRE_LIST Table function (in SQL Tools licensed program) that returns a list of the remote database directories entries you've already setup.

Most SQL iQuery Script users utilize Session Variables on the WHERE clause of a SELECT statement. But they are not limited to the WHERE clause; they can be used anywhere in the SQL statement to embed portions of the statement at runtime. For example, the actual file name on a FROM clause may be passed to the statement within a Session Variable, similar to the following:

```
#define &filename = 'ORDHIST'
SELECT * FROM myData.&FileName order by ACCTNO;
```

At runtime, the SQL iQuery Script engine produces the following statement:

```
SELECT * FROM myData.ORDHIST order by ACCTNO;
```

This clearly provides a much simpler way to build a statement dynamically.

As an interpreted language, each statement is expanded by translating each session variable name to its *value* at runtime. This allows programmers to build complex SQL statements that contain runtime components, without the challenges and ugliness of a lengthy and vexing concatenation statement.

In the context of conditional statements, when the statement requires a quoted value, then at runtime it will attempt to quote the session variable's content, otherwise it will appear as an unquoted value. Here are two examples:

```
01)#define &REGION = 'CHICAGO'
02)#define &CODE = 2
03)If (&REGION = 'PHOENIX');
04)IF (&CODE > 0);
```

Line 3 produces a quoted runtime result of: IF ('CHICAGO' = 'PHOENIX') while line 4 produces an unquoted result of: IF (2 > 0).

SQL iQuery Script is context-aware; in most situations it works the way you want it to work. But when in doubt, you can always quote the session variable name in a conditional statement, just to be sure. SQL iQuery Script will not quote a session variable that already has enclosing quotes around it.

Converting QM Query and Query/400 Queries to SQL iQuery

If you have a library of Query/400 queries, you can use the IBM-supplied RTVQMQRY CL command to convert those existing queries to SQL statements. To do that, use the RUNQRMQRY with the name of the Query and the source member where the generated SQL statement should appear. For example:

```
RTVQMQRY QMQRY(BOBSLIB/MYREPORT) SRCFILE(BOBSRC/QSQLSRC) ALWQRYDFN(*YES)
```

This will produce a source member named MYREPORT that contains Query headers and an SQL SELECT statement that reproduces your results with some omissions.

```
01)H QM4 05 Q 01 E V W E R 01 03 23/01/05 07:56
02)V 1001 050 Customer Balance Due Report
03)V 5001 004 *HEX
04)SELECT
05)   ALL       CUSNUM, LSTNAM, INIT, STREET, CITY, STATE,
06)             ZIPCOD, CDTLMT, CHGCOD,
07)             (BALDUE), CDTDUE
08)   FROM      QIWS/QCUSTCDT T01
09)   WHERE     BALDUE > 0
```

SQL iQuery can directly read and process this source code, unchanged. It uses the embedded Report Title (line 2) and then processes the SQL SELECT statement on lines 4 to 8.

Note the unusual parens around the BALDUE column in the SELECT clause. This is an indication that some type of function was used on that column. You will need to go into the Query/400 environment and see what that function was because the RTVQMQRY command does not return it.

The only way to know what this parenthetical expression was, is to look at the printed query definition. Sadly, that means launching WKRQRY and issuing a Print Definition on the query itself. Once you do that, scroll down to the Report column formatting and summary functions, and look for anything marked under the "Summary Functions" column:

```
1=SUM(), 2 = AVG(), 3=MIN(), 4=MAX(), and 5=COUNT()
```

In my example, the value is 1 next to the BALDUE column. Since this is not a summary report, that is each record is included where a balance due is greater than 0, I need to use the SQL iQuery #COLTOTALS directive instead of the SQL SUM function. I insert that directive into the source member as follows (line 4):

```
01)H QM4 05 Q 01 E V W E R 01 03 23/01/05 07:56
02)V 1001 050 Customer Balance Due Report
03)V 5001 004 *HEX
04)#COLTOTALS 10
05)SELECT
06)   ALL       CUSNUM, LSTNAM, INIT, STREET, CITY, STATE,
07)             ZIPCOD, CDTLMT, CHGCOD,
08)             (BALDUE), CDTDUE
09)   FROM      QIWS/QCUSTCDT T01
10)   WHERE     BALDUE > 0
```

Line 4, above, contains #COLTOTALS 10. This is an SQL iQuery Script directive or *command* that identifies the relative column number to be accumulated. The BALDUE and CRTDUE columns would appear in printed output with BALDUE column total shown below it; and excerpt of this output appears below:

```
  BALDUE    CDTDUE
  101.00       .00
  439.00       .00
 3987.50    300.00
  250.00    100.00
  490.50  -1234.50
  500.00       .00
 5768.00  ***
```

While #COLTOTALS supports both relative column number (10 in this example) as well as the column name (e.g. #COLTOTALS BALDUE) I prefer to use the relative column number since the name of columns is often obscured or lost while processing an SQL statement. For example, if BALDUE had something like a column heading on it, (i.e., the AS clause) then it is no longer known as BALDUE. Likewise, if you cast a column to another length or type, it loses its original name.

You quickly learn to replace the "V 1001" header lines generated by RTVQMQRY with the #Hx directives. For example, lines 1 and 3 are not important, but line 2 contains the "V 1001" value which is the Report Heading. In the above script, lines 1 and 2 can be deleted and line 3 can be replaced with the #H1 directive. That would mean the header is the only line needed, and is replaced with #H1 as follows:

```
#H1 Customer Balance Due Report
```

Now the SQL iQuery Script version of that Query/400 source looks like the following:

```
01)#H1 Customer Balance Due Report
02)#COLTOTALS 10
03)SELECT
04)         CUSNUM, LSTNAM, INIT, STREET, CITY, STATE,
05)         ZIPCOD, CDTLMT, CHGCOD,
06)         BALDUE, CDTDUE
07)  FROM   QIWS/QCUSTCDT T01
08)  WHERE  BALDUE > 0;
```

You'll note that I've removed the ALL clause from the SELECT as it is redundant and often confuses users, I've also removed the parens from around the BALDUE column. Lastly I've added a semi-colon to terminate the statement.

SQL IQUERY SCRIPT

The basic SQL iQuery Script source member is structured with 1 to 3 sections. Advanced scripts can have more sections, may include other source members, and communicate with the Web Browser through the CGI interface. But an SQL iQuery Script typically has either of the following structures:

Basic Script:

> *SQL Statement 1;*
> *SQL Statement 2;*
> *SQL Statement 3;*

Classic Script:

> *Output Headers*
>
> *Declare Session Variables*
>
> *Conditional Logic*
>
> *SQL Statements...*
>
> *Final Statement*

The point of the above illustration is to show that an SQL iQuery Script may contain just SQL statement(s) or it can be an entire application. You can even avoid doing any SQL statements entirely and instead, write stuff to the joblog, run CL commands, and retrieve system-related information (sysvals, serial number, etc.). For example, you may want to test a formula or a regular expression using iQuery Script and write the results to the joblog for review.

Let's look at examples of each sections of an SQL iQuery Script.

Basic iQuery Script

If you have an elaborate SQL statement (perhaps one that spans many source lines) you may want to keep it in a source member so it can be easily run. This is the simplest form of an iQuery Script. You simply store that SQL statement in the source member, and then when you need to run it, you use:

```
RUNiQRY SRCFILE(mysrc/qSQLSRC) SRCMBR(CUSTLIST)
```

This runs the SQL statement(s) stored in the CUSTLIST source member. If you want to print the results instead of viewing them on the display, add the OUTPUT(*PRINT) parameter to the above RUNiQRY command.

Classic Script

While an SQL iQuery Script can be a little as one line/one statement, most scripts have multiple lines of code, including #Hx directives, Session Variables, and conditional logic. All of that is normally followed by an SQL statement that produces the desired result. That final statement may be virtually any SQL statement but it is typically the SELECT statement.

Output Headers

SQL iQuery supports customer headings. These can be thought of as Report Titles and may be specified using the #Hx directive or specified on one of the title parameters of the RUNiQRY command itself. I tend to use the #Hx directives exclusively.

```
#Hx <output title text char(50)>
```

Each #Hx directive may contain up to 50 characters of text. The text should not be quoted, but if it is, those quotes will appear in the heading result. Everything through the end of the line or the first 50 characters is used. #Hx statement do not need to be specified in sequence as the x in #Hx identifies the sequence of the heading line.

Here is an example:

```
01)#H1 Cozzi Productions, Inc.
02)#h2 New York Regional Report
03)#default &REGION 'NY'
04)Select * from qiws.qcustcdt WHERE STATE = '&REGION';
```

Lines 1 and 2 use the #Hx header commands to indicate the text that appears on lines 1 and 2 of the output. Headers are used for many output media types. The display/interactive output supports up to 3 header lines (i.e., #H1 #H2 and #H3) while other output types such as *PRINT, *PDF, *EXCEL also support a fourth header line (#H4) giving you that extra line of headers when needed.

Declaring Session Variables

Session Variables may be declared or deleted. There two directives that may be used to explicitly define a Session Variable and another that may be used to delete or "undefine" a session variable. In addition, Session Variables may be declared on the SETVAR parameter of the RUNiQRY command, or implicitly defined on an EVAL opcode or SELECT INTO or VALUE INTO statements.

Session variables may be passed into the declared within the SQL iQuery Script or passed to the script via the SETVAR parameter of the RUNiQRY command. When passed in on SETVAR a session variable has the same characteristics as using the #define directive in the SQL iQuery Script.

In SQL iQuery Script, the 3 directives that are used to declare or destroy a Session Variable are:

- #define – Define a Session Variable, optionally assign it a value.
- #default – If the Session Variable does not exist, define it and assign it a value.
- #undef – Undefine or *delete* a Session Variable from memory.

Note that when a session variable is passed to a script via the SETVAR parameter of the RUNiQRY command, if a #define directive is specified for the same variable name, the #define directive take priority and replaces that Session Variable. When you need the SETVAR value to be used when specified, then use the #default directive instead of the #define directive.

For more information see the table of Directives later in this document.

Conditional Logic

SQL iQuery Scripts has a basic set of conditional logic opcodes. They allow you to control the flow of the script. Conditional statements may be nested and may control the execution of SQL statements, such as INSERT, UPDATE, SELECT, etc. Conditional statements must be terminated with a semicolon, similar to RPG IV.

As with all most programming languages, conditional statements may span multiple source lines when necessary; the statement continues until a semicolon terminator character is detected. Note that there are # directive versions of these opcodes that have a syntax similar to other directives. Those # directive versions may be used to conditionally build an SQL statement at runtime. More on this feature, later.

OpCode	Description
IF <condition>	Standard conditional statement. Supports all SQL syntax that may appear on a standard SQL WHERE clause. For example: IF (SUBSTR(&PRODUCT,3,4) = 'DEFG'); All conditional statements must be terminated with a semicolon.
IF EXISTS (select statement...);	The IF EXISTS *select-stmt* test can be used with a nested SELECT statement to determine if any rows exist before continuing. IF EXISTS (SELECT ...); -- conditioned code goes here endif;
If EXIST <object> objtype;	The IF EXISTS library/object *objtype; test can be used to check if an IBM i object or member exists. To check for object existance: IF EXIST libname/objname *objtype; To test if a member exists in a file: IF EXIST srclib/srcfName(mbrName) *MBR;
ELSEIF	A combination of ELSE and IF. When the prior IF or ELSEIF condition is false, the next ELSEIF is tested. This continues until an ELSEIF condition is true or the corresponding ENDIF is encountered.
ELSE	When the above ELSEIF or IF (when no ELSEIF conditions are specified) fails, the one-and-only final ELSE statement receives control and any statements following it are performed until the corresponding ENDIF statement is encounter.
DO FOR	Classic "for" loop that auto-increments a counter. It has syntax similar to RPG IV. The opcode is FOR or DO (either can be used). FOR &i = 1 to 10; -- statements go here endfor; Users may use the DO opcode as a synonym the FOR opcode.

WHILE	Classic "do while" loop. Performs the code while the condition is met.
	WHILE (&COUNT < &LIMIT);
	-- your while-loop code goes here
	endwhile;

OpCode	Description

FOREACH
FOR EACH
	Processes each resultSet row of an embedded SELECT statement, returning each row's data to the Session Variables on the INTO clause. Once EOF is reached, the FOREACH loop terminates and closes the SQL Cursor.
	FOREACH select cusnum,baldue
	INTO :CustNo, :Due
	FROM QIWS.QCUSTCDT
	WHERE STATE = 'NY';
	If (&Due > 0);
	#MSG Customer &CUSTNO has a balance due of &Due
	endIf;
	endfor;

LEAVE BREAK	The LEAVE or BREAK opcodes exit the current Loop (FOR, DO, FOREACH, WHILE) immediately and jump to the line after the corresponding ENDDO/ENDFOR/ENDWHILE statement.
ITER ITERATE CONTINUE	Logic is passed to the top of the current loop block.

END IF
END FOR
END DO
END WHILE
	Each conditional statement must be closed with a corresponding ENDxxx statement. Note that the END opcode syntax is flexible in that it supports both the RPG IV style and SQL/PL style. That is, both of the following are supported:
	END IF; or ENDIF;

The IF statement supports the EXISTS extension, as mentioned. It allows the IF EXISTS opcode to be followed by an SQL select statement that when it produces a result, the condition is considered TRUE. If no results are returned, the condition is FALSE. The IF EXISTS is followed by the SQL SELECT statement enclosed in parentheses.

```
if exists (select * from qiws.qcustcdt where BALDUE > 0);
```

You may also specify it with the NOT operator:

```
if NOT exists (select * from qiws.qcustcdt where BALDUE > 0);
```

In addition, IF EXISTS may be used to test for the existence of an IBM i object. To do that, simply specify the qualified object name followed by the IBM i object type. For example:

```
if exists qiws.qcustcdt *file;
```

The IF NOT EXISTS may also be used; for example:

```
if NOT exists qiws/qcustcdt *file;
```

You can see from this last example that either qualifier symbol for object names may be used. That is: *lib/object* and *lib.object* are supported.

When a multi-member file is being checked, you can specify it as illustrated above, or you can include the member's name to verify that a specific member exists in the file. The member exists syntax is illustrated below on line 1:

```
01) IF NOT EXISTS devsrc/qsqlsrc(THX1138) *MBR;
02)     CL: addpfm devsrc/qsqlsrc MBR(THX1138) SRCTYPE(SQL);
03) endif;
```

I particularly enjoy using IF EXISTS to check the existence of data areas which I use as flow control or to enable features I've coded into the script itself. For example, I can test to see if the data area named IQDEBUG exists in QGPL, and then route the code path as desired.

```
If exists qgpl/IQDEBUG *DTAARA;
   #MSG Debug Mode Detected. Dumping Session Variables
   #dumpVars Dev Mode
endif;
```

In the above example, I check if the data area IQDEBUG exists in the QGPL library. I can omit the library and *LIBL would be used. If it exists, I write a message to the joblog and then dump out all existing Session Variable names and their contents. The #DUMPVARS direcctive writes the name and content of all session variables to the joblog.

SQL Statements

Intermixed with conditional logic or as stand-alone statements, virtually any SQL statement may be used within the SQL iQuery Script. This includes but is not limited to: SELECT INTO, VALUES INTO, INSERT, UPDATE, DELETE, DROP, CREATE, DECLARE GLOBAL TEMPORARY TABLE, MERGE and so on. The statements are processed by first translating any embedded Session Variables into their corresponding values, then they are sent to the SQL CLI engine for processing using native IBM i interfaces.

The only session variables that are not translated are those on the INTO clause of a SELECT or VALUES statement. Their value is assigned by the SQL statement itself.

Final Statement

When the final statement in an SQL iQuery Script is any SQL statement, normally it is a SELECT or VALUES statement but can be any SQL statement, that statement is passed back to the RUNiQRY program for processing. That is, the SQL iQuery Script processor does not run that final statement, but instead, builds it and then returns it to the RUNiQRY program. Since it isn't processed by SQL iQuery Script, the INTO clause is not permitted on the final SQL statement.

SQL iQuery Script Components

SQL iQuery Script not only supports session variables, but it also supports:

- Figurative constants
- Directives
- Opcodes
- Built-in Functions

Each of these are explain in the following sections.

Figurative Constants

As with any programming language SQL iQuery Script includes a set of predefined values, which we call Figurative Constants. There are a number of SQL iQuery figurative constants, including:

Constant	Name	Description
*DATE	Current System Date	The current date in job date format. To alter the format of this result, use the CHGJOB DATFMT() command.
*TIME	Current Time	
*DAY	Current Day of the Week	The full day of the week name, e.g., Sunday, Monday, etc.
*SRCMBR	iQuery Script source member name	The name of the source member where the *SRCMBR appears.
*SYSNAME	The Partition ID *system name*.	The system name or partition ID. This is materialized using the MATMATR MI instruction, but is the same name returned by the RTVNETA CL command.
*SRLNBR	The Partition Serial Number	The IBM i Partition serial number. The MATMATR MI instruction is used to extract this data, however it is the same info returned by the QSRLNBR system value.
*USRPRF	User profile	The IBM i user profile of the user that launched the SQL iQuery Script.
*CURUSR	Current User profile	The current user profile under which the job is running

Constant	Name	Description
*GRPPRF	Group Profile	The Group User Profile under which the job is running.
*JOB_NAME	Qualified 3-part job name	The job identifier in 3-part qualified job name format. *nnnnnn/uuuuuuuuuu/jjjjjjjjj*
*JOBNAME	The 10 character name component of the job name associated with the job running the SQL iQuery script.	The job name component of the job identifier.
*JOBUSER	The 10 character user profile associated with the job running the SQL iQuery script.	The user profile that started the job.
*JOBNBR	The 6-character job number associated with the job running the SQL iQuery script.	The, up to, 6-digit job number of the job.
*JOBDATE	The job date	The job date returned in CYYMMDD format. Note the job date is modified when the CHGJOB DATE(...) command is run. Otherwise, it remains unchanged regardless of the duration of the job.
*CCSID	Job CCSID	The job's CCSID under which the job is running.
*DFTCCSID	The Job's default CCSID	The default CCSID if the job is running under a "no CCSID" configuration. For example, the job's CCSID is 65535.
*SYSCCSID	The value of the QCCSID System value	The system value QCCSID value.
*HOMEDIR	User's home folder	The home directory associated with the user profile running the script.
*PRODLIB	SQL iQuery installation Library name	The 10-character library name were SQL iQuery was installed. Normally this is IQUERY and cannot be changed.

*VxRyMz
*VxRy
		Returns true if the IBM i operating system version and release level is at least at that specified. For example:
		if defined(*V7R5);
		...
		endif;
		The above condition is true if IBM i V7R5 or later is installed. This returns true when V7R5, V7R5, V7R8, V8R1 etc. have been installed, but fails (returns false) if the IBM i release level is V7R4 or earlier.

*WEBUSER	Remote User for CGI/Web Brower user.	When running iQuery for Web, this constant is converted to the user profile who signed into the web browser through the secured interface.

These figurative constants may be specified anywhere a regular Session Variable is allowed; except they may NOT be used as part of an SQL statement.

If a figurative constant value needs to be used in an SQL statement, assign its value to a Session Variable, and then use that session variable in the SQL statement.

In the context of the *VxRy and *VxRyMz constants, when used on conditional statements, wrap them in the defined() or isDefined() built-in functions to provide a true/false conditional test. For example:

```
IF defined(*V7R4);
```

iQuery Script Directives

There a several non-conditional, non-SQL operators which we call iQuery Script Directives. They have also been referred to as *hashtag commands* by our users, so either term is acceptable. You've already read about #H1, #define and a few others. Directives occupy a single line of code, typically, and perform one task, such as assigning the output media, setting up the headings, or controlling certain attributes.

There are several directives, the related to programing SQL iQuery Scripts are included in the table below. There are many others that will be covered in context as you progress through this document.

Directive	Parameters	Description
#define	Variable name Value	Declare a variable and assign a value to that variable. #define &count 1 NOTE: #Declare may be used as a synonym of the #define directive.
#default	Variable name Default value	If a variable does not exist, declare the variable and assign it the specified value. If it does exist, ignore the statement. #define &limit 100
#sysval	Variable name System Value	Retrieve system value. #sysval &DATFMT = QDATFMT Note that the GETSYSVAL() built-in function was also introduced in iQuery V5 and may be used in place of the #sysval directive.

Directive	Parameters	Description
#Hx	Title text	Declare output title line. #H1 to #H3 are supported by all output formats, #H4 to #H9 is also supported by Excel output.
#sndmsg	Message text	Send an INFO message to the joblog
#sndsts	Message text	Send a *STATUS message to the *EXT message queue.
#snddiag	Message text	Send a *DIAG message to the job's message queue.
#dumpVars	Title	Writes the name of each Session Variable along with its contents to the joblog as an *INFO message. Note the command has two spellings: #DUMPVAR or #DUMPVARS

Directives are also referred to as *hashtag commands*, and have the following syntax:

```
#XXXXXXXXXX  <value>
```

In the rare case where you need more space for the value specified for a Directive, you may specify a trailing back slash symbol \ and the iQuery Script processor will continue the directive onto the next line, swallowing the backslash itself. The first non-blank character on the subsequent line is inserted where the backslash was located. For example:

```
#msg Sometimes a message can exceed the length of the source line. \
When it does, adding a backslash to the end will continue that \
Statement onto the next line.
```

SQL iQuery Script Built-in Functions

There are a number of built-in functions available to assignment statements (EVAL opcode) and conditional logic in iQuery Script. They help with typical programming tasks such as locating data in a variable, retrieving information, or transforming data. Most are performed inline, while others are independent functions that are run and their value returned.

Built-in	Description
sst(*value, start, len*)	The classic Substring function. The SST function extracts a substring value and passes it to the resulting parser. If you use the native SQL SUBSTR function instead, then it itself is passed to the parser and not processed by SQL iQuery Scrdipt.

Built-in	Description
findfirstOf(*'character list', variable*) firstOf find_first_of	Returns the location of the first position in the 2nd parameter of any of the characters specified on the first parameter.
findfirstNotOf(*'character list', variable*) firstNotOf find_first_not_of	Returns the location of the first position in the 2nd parameter that does not match all of the characters specified on the first parameter.
getCookie(*'cookie name'*)	Retrieve a Web/CGI Cookie of the name specified. The entire value of the cookie is retrieved.
getCWD() getCurDir()	Retrieve the current directory for the job. The result is the IFS path that the job is using as the CWD or *current working directory*.
getHomeDir()	Retrieve the current user's home folder/directory.
trimRight(*'value', length or character*)	Trim the character(s) (specified on the 2nd parameter) from the right-end of the value specified on the first parameter. Optionally, you may truncate a value by specify are hard-coded length. For example, if a value is 200 bytes long and you want the left-most 100 bytes, you can truncate it with: trimRight(&var, 100) If only one parameter is specified, then blanks are removed.
trimLeft(*'value', 'chacter(s)'*)	Trim the leading (left side) characters (specified on the 2nd parameter) from the value specified on the first parameter. If only one parameter is specified, then blanks are removed.
usrspc(*qualified_user_space, start, length)*	Retrieve data from the qualified User Space (*USRSPC) name (first parm). The start positions (parameter 2) and optional length (parameter 3) may be specified to refine the location of the data be retrieved. If the start is omitted the entire user space content is retrieved. If the length is omitted, then the data returned is from the start location through the end of the user space.
getsysval(*'system value'*)	Retrieve System Value. The system value specified on the first parameter is returned.
getsrlnbr*()*	Retrieve the system serial number
getenv*('env variable')*	Retrieve a Job-level Environment Variable's value
getsysenv('*env variable'*)	Retrieve a System-level Environment Variable's value.
msgid*(msgid, msgfile, msgdata, 1 | 2)*	Retrieve message text (MSGID) returns the 1st or 2nd-level message text for the specified MSGID. It inserts the message data (3rd parameter) into the result. If no message data is needed but the 2nd-level text is desired, pass a quoted string/blank for the 3rd parameter and 2 for the fourth parameter. The message ID is a 7-position quoted message ID. The message file (MSGFILE) parameter is a qualified or unqualified *MSGF object name, such as 'QSYS/QCPFMSG' or simply 'QCPFMSG'. `&warning = msgid('CPF3741','qcpfmsg');`
getfile*('ifs stream file name')*	Read contents of IFS file, substituting any embedded Session Variables. The getfile() function reads an IFS file and scans it for any embedded Session Variables. If it detect them it replaces those session variable names with the content of the session variable. This is helpful for things like email/mail-merge applications or HTML loading where you want to populate the HTML "template" with certain data.

Built-in	Description
Fkey*(keyID)*	Checks the environment to see if the corresponding Fn key was pressed to return control to the Script. This is used by SQL iQuery Prompter only, and does not function with other interfaces.
toAscii*(value)*	Converts the value from the job CCSID to CCSID 1208 and returns it to the target session variable.
toEBCDIC*(value)*	Converts the value from ASCII to the job's CCSID, such as 37 for North America. The conversion is based on the CCSID of the JOB when SQL iQuery was started.
dtaara(*qualified_data_area, start, length)*	Retrieve data from the qualified Data Area. The first parameter is the data area name (qualified or unqualified) followed by (2nd parameter) the starting position within the data area and the number of bytes to retrieve (3rd parameter). If the start is omitted the entire data area is retrieved. If the length is omitted, then the data returned is from the start location through the end of the data area.
CompNoCase(*value1, value2*)	Compares the two parameter values using case-insensitive logic. Returns true of a match is detected, otherwise it returns false.
scan(*pattern, searched-value*) scani(*pattern, searched-value*) find(*pattern, searched-value*) findi(*pattern, searched-value*)	The scan and find functions scan parameter 2 for the value specified on parameter 1 and return the location if found, or 0 if not found. The SCANI and FINDI function do the same thing but using case-insensitive logic.
chkLiblE('*library1 library 2...*', *1 | 0*)	The CHKLIBLE function scans the library list of the job for the library/libraries specified and if found, returns a non-zero value. When parameter 2 is specified as 1, (find all) and multiple library names are specified on parameter 1, then all of the library names must exist or 0 is returned.
strlen(*variable*)	The STRLEN function returns the length of the content of the Session Variable specified as its one and only parameter. The content length omits trailing blanks on the right-side.
isDDE(*variable*)	After a successful SELECT INTO, a host variable may have had a Decimal Data Error. Use ISDDE to check for if that field received a DDE by the SQL engine.
isNULL(*variable*)	After a successful SELECT INTO, a host variable may have been set to NULL by the SQL engine. Use ISNULL to determine if that host variable was set to NULL.
split(*variable, 'split-characters'*)	The SPLIT function scans the value (1st parameter) for any of the split-characters (2nd parameter) and returns the data in VALUE up to that point. It then reduces the value size by removing the retrieved content. A subsequent SPLIT to the same session variable will return the next value and so on. Use this function spilt-apart or *parse* the contents of the variable. Example: use this to extract each value of a CSV string or to break apart a qualified object or job name.

Built-in	Description
elems(*variable*)	The ELEM or ELEMS function counts the number of array elements in a Session Variable Array (SVA). SVAs are created when a SELECT INTO is used with multiple resultSet rows/values. For example, if the FOR 5 ROWS ONLY or the LIMIT 5 is used, then the target host variables will receive up to 5 values from the resultSet. That causes each of the Session Variables of the INTO clause to be converted into Session Variable Arrays. The individual array elements may be accessed using standard C/C++ array syntax: &varArr[*index*] where *index* is a literal or another session variable.
toUpper(*variable*) toLower(*variable*)	The TOUPPER and TOLOWER functions convert the contents of the session variable (parameter 1) to all upper or all lower case. The input variable (parameter 1) is not altered by these functions. e.g., IF (toLower(&part) = 'sdm');
xLate(*'from-characters', 'to-characters', variable*)	The XLATE function is similar to the RPGIV %XLATE built-in function; it translates each character from the from-characters (parameter 1) found in the variable (parameter 3) to the corresponding character of the to-characters (parameter 3).
encodeXML(*variable*)	The ENCODEXML function escapes each of the 5 XML reserved symbolic characters to the HTML escape pattern. For example, the & (ampersand) symbol is translated to & The 5 symbols required by XML to be escaped, include: & , " , ' , < , >
encodeURL(*variable*)	The ENCODEURL function escapes any symbol in the *variable* (parameter 1) that requires it to be escaped. It escapes these characters by converting them to Hex and adding the percent sign (%) prefix. If a blank is encountered, it is converted to a plus sign.

editCode(*variable, edit-code*)
editC(*variable, edit-code*)
	The EDITCODE and EDITC functions edit the numeric value specified on the first parameter using the single-character edit code specified on the second parameter. For example:
	`#define &Credit = -1024.25`
	`Eval &eCredit = editc(&credit, 'J');`
	`Results in &ECREDIT = '1,024.25-'`

editWord(*variable, edit-mask*)
editW(*variable, edit-mask*)
	The EDITWORD and EDITW functions edit the numeric value using the edit mask specified on the 2nd parameter.
	`#define &SALES = 1200.50`
	`&eSales = editw(&sales, '$ , 0 . -');`
	`Results in &ESALES = $1,200.50-`

isEmpty(*variable*) isNotEmpty(*variable*)	The ISEMPTY function returns true if the session variable specified on parameter 1 either does not exist or exists and contains no data or contains only blanks ("is empty"). The ISNOTEMPTY function returns true if the session variable exists and contains data.
isBatch()	Returns true if the SQL iQuery Script is being run as a batch job on the system.
isInter() isInteract()	Returns true if the SQL iQuery Script is being run in an Interactive job.
isWeb()	Returns true if the SQL iQuery Script is being run as an HTTP job from a CGI call from a web browser or similar.

Built-in	Description
isBrowser()	Returns the HTTP User Agent that evoked the SQL iQuery script as a CGI job. The web browser information is returned. Use this function to copy that information into a session variable or to scan for specific browser Authors, such as WebKit, Microsoft, etc.
isNum(*variable*)	Returns true if the session variable contains a numeric values, 0 to 9, decimal notation, status (i.e., the minus sign) and a thousands separator.
isDigits(*variable*)	Returns true if the session variable contains all numeric digits 0 to 9 only.
isAlpha(*variable*)	Returns true if the session variable contains only alphabetic characters.
isChar(*variable*)	Returns true if the session variable contains any alphabetic characters, numbers (0 to 9) or blanks. If any symbols are detected, such as % # @, etc. it returns false.
isAlphaNum(*variable*)	Returns true if the session variable contains any alphabetic characters or any digits (0 to 9).

toHex(*variable*)	Returns the 2-character hexadecimal representation for each single character in the session variable (parameter 1). For example:
	`Eval &hexVal = toHex('12345');`
	`Results in &HEXVAL = 'F1F2F3F4F5'`

checkObjType(*variable*)	Returns true if the IBM i object type in parameter 1 is a valid object type. It does this by attempting to convert the symbolic/external object type, such as *FILE or *PGM, to the internal MI object type. If that conversion fails, the object type is considered invalid. This provides the best support for future IBM i versions that may contain new/additional object types.

chkObjExists(*object, object-type*)	Returns true if the IBM i object (parameter 1) of the type specified on parameter 2 exists. For example, to check if the object QCUSTCDT exists use this statement:
	`IF chkObjExist('qiws/qcustcdt',*FILE);`
	To check to see if a specific source member in a file exists, use *MBR as the object type and encose the member name in quotes:
	`IF chkObjExist('coztools/qrpglesrc(cpytocsv)',*MBR);`

cpybytes(*target var, source var*)	The CPYBYTES function copies the content of the source variable (parameter 2) to the target variable (parameter 1) directly. Normally a simple eval/assignment is used to copy session variables. However, when a complex session variable, such as an array or LOB (when it contains the content of an IFS file), then it is sometimes easier to use the CPYBYTES function to replicate the variable entirely.

Built-in	Description
Print BEFORE | AFTER (*start-column, argument1, arg2...*);	The PRINT command may be used to write addition text information before or after the Result Set. It is often used to add additional information after the list of rows that a SELECT statement returns. Use the PRINT command followed by the AFTER() or the BEFORE() keyword. Then within the parens, specify the starting column in the result followed by the text to be added. Note that this was originally created for EXCEL-based output, but now also works with printed output. `PRINT after(3,'Confidential. Do not distribute');` This prints the text after the result set, starting in column 3 (column C in Excel). See PRINT Command elsewhere in the document for more information and additional examples.

In addition to this list of built-in functions, all available scalar SQL built-in functions as well an any add-on scalar SQL functions may be used in SQL iQuery control statements. They may not be used on the EVAL assignment statement, but may be used on VALUES INTO instead of the EVAL opcode. For example, the licensed program named *SQL Tools* (2COZ-ST1) ships with an EOMDATE scalar function that returns the end of month date. It can be used directly on an IF statement, as follows:

```
01) IF (current_date = sqlTools.EOMdate());
02)   Include prodlib/qsqlsrc(monthEnd);
03) else;
04)   Include prodlib/qsqlsrc(daily);
05) endif;
```

Line 1 illustrates the scalar function EOMDATE (an *SQL Tools* function). If the current date is equal to the end-of-month date, then the MONTHEND SQL iQuery Script source member is included. If it isn't month-end, then the DAILY source member is included.

SQL iQuery Script Commands

SQL iQuery Script support several commands or *opcodes* to perform various tasks. Commands are terminated with a semicolon and may span multiple lines.

Command	Description
EVAL &var = *value or expression*	The EVAL opcode is the classic assignment statement in SQL iQuery Script. The left side of the equals sign is the target variable, while the right-side contains either another variable, a literal/string, or any mathematical expression such a &A * &B, or any SQL scalar function. The EVAL opcode itself is optional as of SQL iQuery version 7. In some conditions the SQL VALUES INTO statement may be better suited than the built-in EVAL opcode, but that's a decision for the specific circumstances.
return;	The RETURN opcode returns to the "caller". That is it ends the current SQL iQuery script and returns to the prior script (when using nested scripts) or returns to the RUNiQRY command processing program. Effectively ending the current script immediately.

ftp <ftp command> <parameters>;	The FTP opcode can be used to setup a list of objects/files to be sent via FTP and then send those objects. The maximum number of source lines generated to an FTP script cannot exceed 32760.
	The parameters are standard FTP commands and their parameter values. For example, to create a directory on a target system, you might specify:
	`FTP MK /home/query;`
	To send a file:
	`FTP CD /home/query;`
	`FTP PUT /home/workfolder/mydata.txt;`
	More on this topic is coming in a future document.
	Then to run the script:
	`FTP connect CHICAGO FTPUSER 'rosebud';`
	`FTP RUN;`
	More on this topic is coming in a future document update.

CONNECT to <rmt> user <user> using <pwd>;	The classic SQL CONNECT TO statement is integrated into iQuery script. Specify it as you would any other CONNECT TO. Note that to reset the connection you use CONNECT RESET;
savestmf <ifs file>, *variable*, ccsid;	Saves the session variable content to the IFS file. If the IFS file does not exist, it is created using the CCSID parameter's CCSID. If no CCSID parameter is specified it defaults to CCSID(1208). If the IFS file already exists, it is cleared, then the session variable's content replaces any existing data. Use the APPENDSTMF command to add data to an existing IFS file.
appendStmf <ifs file>, *variable*, ccsid;	Writes the content of the session variable to the end of an existing stream file. If the stream file does not exist, it is created.
writeStmf <ifs file>, *variable*, ccsid;	Writes the content of the session variable to the stream file. If the stream file exits, the data is added to the end of the file. If the file does not exist, it is created first, and then the data is written to it.
PRINT BEFORE(column, data...)	Write the data, starting in the column specified, above the output resultSet. Valid for print and Excel output only.
PRINT AFTER(column, data...)	Write the data, starting in the column specified, below the output resultSet. Valid for print and Excel output only.

The PRINT Command

The PRINT command provides supports to include additional text in the output. Originally written to provide a way to embed information after the resultset rows in EXCEL output, it was expanded to work with printed output as well.

Syntax:

```
PRINT after( <starting-column>, output-text,output-text2...);
```

The PRINT command supports the AFTER or BEFORE keywords. When BEFORE is used the text appears before (above) the result set, when AFTER is used, the text appears below (after) the result set.

Parameter 1 must contain a number greater than 0 that represents the starting EXCEL column into which the text specified on parameter 2 is written.

Parameter 2 and beyond contain quoted text strings that are written to the output starting at the column specified in parameter 1. Each time a parameter marker is encountered (parameter marker is a comma) the text in the next parameter is written out to the subsequent column. For example, below the works, RED, WHITE and BLUE are written to columns 3, 4 and 5 respectively:

```
PRINT AFTER(3,'RED','WHITE','BLUE');
```

In place of text parameters, you may also specify an EXCEL-based Style for the text. The STYLE keyword is used to accomplish this. For example. Suppose you wanted the output text to be BLUE instead of the default color. The STYLE keyword along with the Excel formatting code may be used as follows:

```
PRINT AFTER(3, style( color: blue), 'Hello World');
```

The *style* keyword accepts MS Excel style attributes using a format substantially similar to that of HTML CSS. That is the Attribute name followed by a colon followed by the attribute. Multiple attributes are separated from one another with a comma.

```
PRINT AFTER(3, style( color:blue, rotate: 45,fontsize:18), 'Hello World');
```

It is up to the user to know the Excel styles and their syntax. Embedding an incorrect style or using the wrong style format will result in Excel issuing an error and potentially failing to open the Spreadsheet. Be sure to test the scripts before providing end-users with the results.

CODING EXAMPLES

When creating an SQL script, you often need to test the results of the prior SQL statement. SQL iQuery Script supports the &SQLSTATE session variable. It can be used to test the SQL state after each SQL statement. For example, if you wanted to extract the operating system's technology refresh level.

```
01)  -- Get IBM i Technology Refresh Level into &TR
02)SELECT  PTF_GROUP_LEVEL
03) INTO :TR
04)FROM QSYS2.GROUP_PTF_INFO
05) WHERE PTF_GROUP_DESCRIPTION = 'TECHNOLOGY REFRESH'
06)     AND PTF_GROUP_STATUS = 'INSTALLED'
07)ORDER BY PTF_GROUP_TARGET_RELEASE DESC,PTF_GROUP_LEVEL DESC
08)LIMIT 1;
09)
10)if (&SQLState >= '02000' or isEmpty(&TR));
11)  eval &TR = 0;
12)endif;
```

The SQL SELECT statement that begins on line 2 and continues through 8 attempts to retrieve the IBM i operation system *technology refresh* level. On line 10, &SQLSTATE is checked for the infamous '02000' state. If &SQLSTATE is 02000 or higher, then the assumption is that no TR level was retrieved, and the &TR session variable is set to 0.

The next step in this script is to check that the OS release level and &TR level are at least at level 9 (for V7R3) or 3 (for V7R4) variable for something useful. In our case, we want to check the TR level of the system before attempt to include the PROTECTION_STATUS column from the SYSDISKSTAT View. IBM added PROTECTION_STATUS late in V7R4 and backported it to V7R3. To check that, an #IF statement directly inserted into the SELECT query and controls the inclusion of the PROTECTION_STATUS column, as follows:

```
13) SELECT ASP_NUMBER as   "ASP",
14)         UNITNBR    as   "Drive               Number",
15)         DISK_TYPE  as   "Disk                Unit Type",
16)         DISK_MODEL as   "Disk                Model",
17)         CASE WHEN UNIT_TYPE = 1 THEN 'SSD'
18)              WHEN UNIT_TYPE = 0 THEN 'HDD'
19)              ELSE '???' END
20)                    as   "Drive Type          HDD/SSD",
21)         UNITMCAP   as   "Drive               Capacity",
22)         UNITSPACE  as   "Drive               Free Space",
23)         PERCENTUSE as   "Percentage          Used",
24)
25) #IF defined(*V7R5) or \
26)     (defined(*V7R3) and &TR >= 9) or \
27)     (defined(*V7R4) and &TR >= 3)
28)         PROTECTION_STATUS
29) #else
30)         'Feature NOT Avail'
31) #endif
32)                    as   "Status",
33)         CASE WHEN MIRRORPS is NULL THEN 'NOT Mirrored'
34)              WHEN MIRRORPS = '0'   THEN 'Inactive'
35)              WHEN MIRRORPS = '1'   THEN 'Mirrored'
36)              END   as   "Mirrored",
37)         CASE WHEN MIRRORUS is NULL THEN ' '
38)              WHEN MIRRORUS = '1'   THEN 'Paired'
39)              WHEN MIRRORUS = '2'   THEN 'Syncing'
40)              WHEN MIRRORUS = '3'   THEN 'Suspended'
41)         END        AS   "Mirrored            Status"
42)  FROM QSYS2.SYSDISKSTAT;
```

Line 13 begins the SELECT query of the SYSDISKSTAT View (see line 41). Then on line 25 the #IF statement is used to test for the IBM i operation system being on V7R3 TR9, V7R4 TR3 or V7R5 or later. If it is not, then a constant of "Feature Not Avail" is produced instead of the drive protection status.

Remember the # (pound sign which has a contemporary nomenclature of *hashtag*) alters the IF condition so it can be directly embedded in an SQL statement. When and #IF statement is embedded like this, it is used to include or omit portions of the SQL statement. Without the hashtag prefix, the SQL iQuery Script processor would concatenate the IF statement with the SQL statement and produce a syntax error.

Let's look closer at that embedded #IF condition portion of the above script.

```
01)#IF defined(*V7R5) or \
02)    (defined(*V7R3) and &TR >= 9) or \
03)    (defined(*V7R4) and &TR >= 3)
04)        PROTECTION_STATUS
05)#else
06)        'Feature NOT Avail'
07)#endif
```

Line 1 uses the DEFINED() built-in function to check if the figurative constant *V7R5 is defined. If it is, then IBM i V7R5 or later is installed, and we're good. Note that since this condition need a bit more space, I've continued it to lines 2 and 3 using the trailing backslash, as mentioned earlier.

If *V7R5 is not defined, then checks for *V7R3 at TR9 or later, or for *V7R4 and TR3 or later. If either condition is met, then the PROTECTION_STATUS column is included in the resultSet. If none of the conditions are met, then the literal 'Feature NOT Avail' is inserted instead of the PROTECTION_STATUS column.

> **Note**: SQL Tools QDISK View and DISK_LIST Table function return the Protection_Status column starting with V7R2. They may be used as an alternative to the IBM-supplied SYSDISKSTAT View.

Before running the SQL SELECT statement that produces the DISK Drive Status Report, you may want to include header information and change some output attributes. To do that, we could add the following 4 lines of code to the previous SQL iQuery Script.

```
01)#H1 My Company Corp.
02)#H2 *MACRO - Script
03)#H3 Disk Drive Status Report
04)#colEdit  6,7
```

Lines 1, 2, and 3 add Report header lines to the output. Output formats that do not support heading simply ignore the #Hx directives.

- #H1 is the company name – a convention most SQL iQuery Customers have adopted.
- #H2 contains the figurative constant *MACRO that inserts the Source File Member name into the heading.
- #H3 contains the report title.

Line 4 uses the #COLEDIT (edit numeric columns) directive to identify the columns that should have standardized numeric editing applied. In this example, the Disk Capacity and Space Available are identified as columns 6 and 7 respectively. You could have also used the column names, but that can be confusing since SQL will use the correlation name as the column name in most situations.

The results of this query would look something like the following: