iSeries SQL Programming: You've Got the Power!

By Thibault Dambrine

On June 6, 1970, Dr. E. F. Codd, an IBM research employee, published "A Relational Model of Data for Large Shared Data Banks," in the Association of Computer Machinery (ACM) journal, *Communications of the ACM*. This landmark paper still to this day defines the relational database model.

The language, Structured English Query Language (known as SEQUEL) was first developed by IBM Corporation, Inc. using Codd's model. SEQUEL later became SQL. Ironically, IBM was not first to bring the first commercial implementation of SQL to market. In 1979, Relational Software, Inc. was, introducing an SQL product named "Oracle".

SQL is the ONLY way to access data on Oracle database systems. The ONLY way to access data on Microsoft SQL Server and I could repeat this sentence with a number of other commercial database products.

On the iSeries however, SQL is NOT the only way to access the data. SQL support has only started with V1R1. At that time, AS/400 SQL performance was a problem – a big turnoff for any developer. Because of these facts, many iSeries developers still consider SQL as an optional skill. Many still use SQL only as an advanced data viewer utility, in effect, ranking SQL somewhere north of DSPPFM.

Today, one can barely get by anymore as a programmer without knowing SQL. The same is true for all commercial database systems. They all support SQL. The iSeries is no different. It has evolved to compete.

As a programming language, SQL is synonymous with

- The ability to do more while coding less than a conventional high-level language like C or RPG
- Efficiency while processing large amounts of records in a short period of time
- A language offering object-oriented like code recycling opportunities, using SQL Functions and SQL Stored Procedures
- Easy to learn and easy to use, SQL is a language that allows one to tell the system what results are required without describing how the data will be extracted

This article's first part will describe some common data manipulation methods using SQL. These are from real life applications and they can be applied to a variety of situations.

The second part will describe ways to implement SQL code, so that it can be called or evoked by a job scheduler just like any other program on your iSeries.

SQL on the iSeries can be implemented in several methods, some of which include using SQL embedded in C, RPG or other languages. In this article however, I will only concentrate only on "pure" SQL, or SQL that is either interpreted (`RUNSQLSTM`) or compiled into a function or a stored procedure.

IBM has packed a lot of power into iSeries SQL. Borrowing Anthony Robbins's words, it is time to "Awaken the Giant Within"!

PART 1: CODING IN SQL

The Data

Unlike most other languages, SQL is not only a language to manipulate data, it is also one that can be used to define the database.

In SQL, the category of statements used for this purpose is called Data Definition Language, or DDL. There are many DDL functions, and the point here is not to go through all of them, but I would like to highlight a few that I find most interesting and useful.

In DDL:

1) You can define both short (usable in old RPG) and more explicit long fields (usable in SQL).
2) You can define date and time fields with the "DATE" and the "TIMESTAMP" types. This allows date manipulation and date calculations in SQL in a far easier way than in conventional languages.
3) Any table defined in SQL is accessible and mapped by the system like any other file created with DDS. You can do a DSPFD or a DSPFFD with such files just as if they had been created with DDS. Indexes are the same, except they are recognized as Logical Files by the system.

Here is an example of this type of definition, with some notes.

```
CREATE   TABLE         ER100F
(
    BATCH_ID               FOR BTCHID        NUMERIC(10)            ,
    SOURCE_FACILITY        FOR SRCFAL        CHAR(50)               ,
    LOAD_TIMESTAMP         FOR LDTMSP        TIMESTAMP
)  ;
LABEL ON ER100F (SOURCE_FACILITY                          TEXT IS
                'Source Facility                                  ');
LABEL ON ER100F (BATCH_ID                                 TEXT IS
                'Batch ID                                         ');
LABEL ON ER100F (LOAD_TIMESTAMP                           TEXT IS
                'Load Timestamp                                   ');

LABEL ON TABLE ER100F IS 'Test Data Fact Table'  ;
```

Note the SOURCE_FACILITY column, defined as CHAR, as opposed to VARCHAR (more common is other SQL implementation). This is because on the iSeries, VARCHAR does cause some extra overhead. If you really need to use VARCHAR, you should know that in this type of field, the first two bytes of the field are used to store the length of the field that is occupied with data. For example, if 39 bytes were busy with data out of the 50 bytes available, the number 39 would be in binary in these two bytes. In the past, I also found that using VARCHAR columns for index keys would best be avoided for performance-critical index columns.

Note the LOAD_TIMESTAMP column. It is defined as TIMESTAMP. This data type is one of the most valuable in SQL, as one can do data calculations, such as the number of days, hours or minutes between two time stamps. There is also a DATE data type, which is equally useful.

Note the "FOR" notation followed by the short field name. This is iSeries-specific and it is there so that the programmer can give a long and a short name to a column. Short here, means 10 characters or less. If you decide to use a long name (greater than 10 characters) for your column without specifying a short name with the FOR operand, the system will create a short name for you. For example: If you chose to use "BATCH_NUMBER" (12 characters in all – which is allowable in SQL), SQL will assign a 10-character short name for your column which will look like "BATCH00001" and is typically not very meaningful.

One more point on creating tables with SQL: To create a table with a source member written like the one above, you have to use the `RUNSQLSTM` command. The table will be created in your CURRENT LIBRARY. If you wish to create it elsewhere, you have to first change the CURRENT LIBRARY for the job first.

Here is an index example: Indexes are visible on the iSeries as Logicals in a DSPDBR command:

```
CREATE UNIQUE INDEX  ER100FIDX  ON ER100F
 (
    BATCH_ID
 )
```

On the index, the long or the short names are both valid to specify keys. Note the "UNIQUE" keyword. This is specifying a unique key and is only needed if you do need the key to be unique. With SQL, on the iSeries, in addition to creating a UNIQUE INDEX, one has also the option to create a UNIQUE CONSTRAINT. Constraints, however, do not show up on the DSPDBR command. To find constraints, one has to look in the SQL Catalog.

Exploring the SQL Catalog

The SQL catalog is a repository of all tables, column names, triggers and generally every component relevant to the database. On the iSeries, one can locate the SQL Catalog tables in library QSYS2. Note that even tables created originally as files with the CRTPF or CRTLF commands will be recorded in the DB2 SQL Catalog tables. These SQL Catalog tables all start with the letters "SYS".

For example, to find the SQL definition of an existing table, you can do a `SELECT * FROM QSYS2/SYSCOLUMNS WHERE TABLE_NAME = 'FILE_NAME'`. Using `SELECT * FROM QSYS2/SYSCHKCST` will show all existing constraints in your system. Other SQL catalog files include SYSINDEXES, SYSFUNCS, SYSTRIGGER and SYSVIEWS.

Critical Nature of Indexes

Often in SQL training sessions, for obvious reasons, the databases used for teaching purposes are small. In these situations, not having an index or not using the best methods produces a result in reasonable time, regardless of the presence or absence of indexes. When processing large amounts of data, absence of an index or using a less than

optimal method (which forces SQL to do more disk access than it would otherwise), will magnify inefficiencies and the time it takes to produce results.

Another point, which cannot be over-emphasized, when putting together a database for use in SQL, is to ensure that the data within is usable "as is". What I mean here is that if you need a date field, use the DATE data type, not a DEC or CHAR. Ensuring that your data is set in the best data type will mean that you will be able to join tables with indexes that have the same type and do not need to be cast or modified to connect.

Altering Tables With Data
SQL has the ability to alter a table, even if it already contains data. Effectively, you can add or remove a column on a table without having to temporarily store the data in a work-file and copying it back after re-compiling the table. Note that the ALTER keyword in SQL can also be used for adding or removing constraints and primary keys.

Adding a column:
```
ALTER TABLE EQP_TABLE ADD COLUMN EQUIPMENT_CATEGORY FOR EQPCAT CHAR(10)
```
Removing a column:
```
ALTER TABLE EQP_TABLE DROP COLUMN EQUIPMENT_CATEGORY
```

Adding a constraint:
```
ALTER TABLE EQP_TABLE ADD CONSTRAINT NEWID UNIQUE(EQP_ID, PURCH_DATE)
```

Using SQL to create Triggers
Another notable SQL feature is its ability to create database triggers. Below, is the syntax for a trigger written with SQL:

Trigger syntax	Trigger Example (re-printed from IBM Manual)
CREATE TRIGGER *trigger_name* *Activation _Time* *Trigger_Event* ON *table_name* REFERENCING OLD AS *old_row* NEW AS *new_row* FOR EACH ROW / FOR EACH STATEMENT MODE DB2ROW / DB2SQL WHEN *condition* BEGIN *Trigger body* END (DB2ROW triggers are activated on each row operation, DB2SQL triggers are activated after all of the row operations have occurred)	CREATE TRIGGER SAL_ADJ AFTER UPDATE OF SALARY ON EMPLOYEE REFERENCING OLD AS OLD_EMP NEW AS NEW_EMP FOR EACH ROW MODE DB2SQL WHEN (NEW_EMP.SALARY > (OLD_EMP.SALARY *1.20)) BEGIN ATOMIC SIGNAL SQLSTATE '75001'('Invalid Salary Increase - Exceeds 20%'); END BEGIN ATOMIC specifies a list of SQL statements that are to be executed for the triggered-action. The statements are executed in the order in which they are specified.

Coding in SQL – Join Basics
One of the must-have building blocks for coding in SQL is the JOIN statement. This is where all the set logic comes alive. While SQL can handle record-by-record processing using cursors, in most cases, it is most efficient when processing sets of records.

Here are the choices we have when joining tables. When two bolded keywords are specified, both syntaxes are correct. One is simply more explicit than the other:

- A **Join** or **Inner Join** returns only the rows from each table that have matching values in the join columns. Any rows that do not have a match between the tables will not appear in the result table.
- A **Left Join or Left Outer Join** returns values for all of the rows from the first table (the table on the left) and the values from the second table for the rows that match. Any rows that do not have a match in the second table will return the null value for all columns from the second table.
- A **Right Join or Right Outer Join** return values for all of the rows from the second table (the table on the right) and the values from the first table for the rows that match. Any rows that do not have a match in the first table will return the null value for all columns from the first table.
- A **Left Exception Join** returns only the rows from the left table that do not have a match in the right table. Columns in the result table that come from the right table have the null value.
- A **Right Exception Join** returns only the rows from the right table that do not have a match in the left table. Columns in the result table that come from the left table have the null value.
- A **Cross Join** returns a row in the result table for each combination of rows from the tables being joined (also known as a Cartesian Product).

The ability to join tables gives the flexibility to assemble data from several sources. It also allows correlations, updates, deletes and verifications.

Bringing SQL to life – Some common coding examples:

Join logic is good to know, but in isolation, it all remains SQL theory. In the following paragraphs, I will describe real life examples and the SQL methods that I have used to resolve these situations.

Example 1: Gathering Information from Several Tables Using a Left Outer Join

Let us assume that your primary file is the employee master, which is keyed by EMPLOYEE_NBR and your secondary files are the BENEFITS_MASTER and the PAYROLL_MASTER. Both also contain an EMPLOYEE_NBR column. These columns will be the join keys. If a BENEFITS_MASTER or PAYROLL_MASTER record cannot be found for a given key in the primary file, SQL by default will return a NULL value. In this example however, we will intercept NULLs with the IFNULL operand and substitute the NULLs with our own defaults.

```
INSERT INTO EMPLOYEE_DATA
(
EMPLOYEE_NBR,
EMPLOYEE_NAME,
EMPLOYEE_BENEFITS_DESC,
EMPLOYEE_SALARY,
SALARY_CATEGORY
)
SELECT
EM.EMPLOYEE_NBR,
EM.EMPLOYEE_FIRST_NAME || ' ' || EM.EMPLOYEE_LAST_NAME,
IFNULL(BM.EMPLOYEE_BENEFITS_DESC, 'New Employee - Benefits not yet allocated'),
IFNULL(PM.YEARLY_SALARY, 0),
CASE
    WHEN PM.YEARLY_SALARY<100000 THEN 'REGULAR EMPLOYEE'
    WHEN PM.YEARLY_SALARY<=100000 THEN 'EXECUTIVE EMPLOYEE'
    WHEN PM.YEARLY_SALARY IS NULL THEN 'UNKNOWN - INVESTIGATE'
    ELSE 'DA BOSS'
END

FROM EMPLOYEE_MASTER EM
       LEFT OUTER JOIN BENEFITS_MASTER BM ON EM.EMPLOYEE_NBR = BM.EMPLOYEE_NBR
       LEFT OUTER JOIN PAYROLL_MASTER PM  ON EM.EMPLOYEE_NBR = PM.EMPLOYEE_NBR;
```

A couple of points regarding the example above:

- If you have more than one match in the outer join (BENEFITS_MASTER in this case) file, SQL will return as many rows as there are matches.
- Left Outer Join is a very efficient model. The example above shows one table (EMPLOYEE_MASTER) joined to a two other tables, the BENEFITS_MASTER and the PAYROLL_MASTER. You can actually add many left outer joins and still get a very efficient retrieval process. In RPG terms, it is similar to putting a read of EMPLOYEE_MASTER in a loop, and for each record, doing a chain to BENEFITS_MASTER and another chain to PAYROLL_MASTER, based upon the returned indicators, putting a default value in the variable and writing the results into EMPLOYEE_DATA.
- Of all the SQL techniques, Left Outer Join is one of the most commonly used to extract data.
- Note the CASE statement used for filling the SALARY_CATEGORY column. CASE statements give your SQL code the flexibility to transform data, as opposed to only gather data as it is. Think of this in a situation where you are using SQL to do a data conversion project. This is one example where using CASE may be most useful.

Example 2: Updating or Deleting Data in a Table Using a Correlated Query

a) An update in a given table, when based on data stored in another table is described in SQL a "correlated update". In this situation, you have to repeat the same "where" clause 2 times. Note the use of correlation names. The main table, FILEX, is correlated to the name "MAIN". The table on which the decision to update is based, FILEY, is correlated to the name UPDT. Giving the correlation names makes each table's role more obvious in the SQL statement. Note also that once the table is given a correlation name, any column that belongs to it must be qualified with that correlation name, as opposed to the original table name, if you chose to qualify it.

```
UPDATE FILEX MAIN
SET (MAIN.FIRST_NAME, MAIN.LAST_NAME) =
(SELECT UPDT.FIRST_NAME,UPDT.LAST_NAME FROM FILEY UPDT WHERE MAIN.ID=UPDT.ID)
WHERE EXISTS
(SELECT 'DUMMY' FROM UPDATE_TABLE UPDT WHERE MAIN.ID=UPDT.ID);
```

b) A delete in one table based on data stored in another table is called "correlated delete". Again, here is an example using the WHERE clause two times:

```
DELETE FROM FILEX MAIN
WHERE EXISTS
(SELECT 'DUMMY' FROM FILEY UPDT WHERE MAIN.ID=UPDT.ID);
```

To formalize the correlation name syntax, one can say it works in the following pattern:
SELECT "TABLE_CORRELATION NAME"."COLUMN_NAME1" "COLUMN_CORRELATION NAME" FROM "TABLE_NAME" "TABLE_CORRELATION NAME"

Example 3: Finding Duplicate Keys in a Table
One of the most common uses for SQL is to identify duplicate values in a raw input table. To find out if there are duplicates, there are several methods, several solutions that you may want to pick from, depending on what you are aiming to do.

a) This example will show how to identify duplicates, based on the criteria of three different columns in your table.

```
SELECT ADDRESS_1, ADDRESS_2, ADDRESS_3, COUNT(*)
FROM CONTACT_TABLE
GROUP BY ADDRESS_1, ADDRESS_2, ADDRESS_3
HAVING COUNT(*) > 1
```

Note the HAVING clause. It applies only on aggregated values, in this case, the COUNT(*). Note: You cannot do a "WHERE COUNT(*) > 1". The WHERE clause is good only for comparing values in individual rows.

b) The example above will show only the key fields of the duplicate rows. If you wish to see the entire row (all the columns), you can use the following method, which joins a table to itself and uses the relative record number, as well as the keys to show the duplicates:

```
SELECT RRN(A1), A1.* FROM CONTACT_TABLE A1
WHERE RRN(A1) < (SELECT MAX(RRN(A2) ) FROM CONTACT_TABLE A2
                 WHERE ( A1.ADDRESS_1 = A2.ADDRESS_1 AND
                         A1.ADDRESS_2 = A2.ADDRESS_2 AND
                         A1.ADDRESS_3 = A2.ADDRESS_3 ) )
ORDER BY A1.ADDRESS_1, A1.ADDRESS_2, A1.ADDRESS_3
```

Note on this example the use of correlation names A1 and A2. Here, the use of correlation names allows us to attack the same table as if they were two distinct objects. One thing to remember, if you use a correlation name for a table, it is wise to use that correlation name to qualify any column you retrieve, especially if you join a table to itself. Another point to remember with the code above is that without index, it can be a time-consuming operation if the file is large.

c) Of course, once duplicates are identified, you may want to remove them. Assuming the ID_NUMBER in this case is the key we want to keep as unique, the code below will remove all the records lower than the greatest key value.

```
DELETE FROM CONTACT_TABLE A1
WHERE A1.ID_NUMBER <
(
    SELECT MAX(A2.ID_NUMBER)
    FROM CONTACT_TABLE A2
    WHERE ( A1.ADDRESS_1 = A2.ADDRESS_1 AND
            A1.ADDRESS_2 = A2.ADDRESS_2 AND
            A1.ADDRESS_3 = A2.ADDRESS_3 )
)
```

d) Extracting Maximum Values where duplicates may be present in a data set:
Let's assume for this example that each batch of cash-register data is time-stamped. Imagine for a second that a cash register was polled twice by accident. Or a situation where half the sales came in, the communication was broken by a power failure and a polling re-try retrieved the entire batch again, thus creating duplicates. Using the time-stamp, you would only be interested in the latest data. Here is what you could do in SQL to get this information:

```
SELECT STOREKEY, CLERK, TRANSACTION_NUMBER, ITEM#, SIZE, COLOUR, DOLLAR_AMT, MAX(POLLING_TIME)
FROM POLLING_TABLE
GROUP BY STOREKEY, CLERK, TRANSACTION_NUMBER, ITEM#, SIZE, COLOUR, DOLLAR_AMT
```

Note that I used the MAX function in the examples above. The MIN function works the same way and returns the smallest value.

Example 4: Joining Two Tables with Incompatible Keys
The solution to this problem may be casting. Using SQL's casting function, you can transform on the fly the contents of a numeric column into characters and vice-versa. With the transformed key, you can then join to the foreign table with a key that has now compatible data types. The only draw-back to this method is that the indexes will not be used. Because of this, on a large scale, it can become an expensive in terms of time and computing resources.

To cast alpha data as numeric, use the following formula:
```
SELECT INT(SUBSTRING(CHARCOLUMNNAME, STARTCHAR, LENGTH)) FROM FILENAME
```
Note that this example also has a substring built-in, meaning we are picking only a portion of column

To cast numeric data as alpha, use the following formula:
```
SELECT CAST(INTEGERCOLUMN AS CHAR(5)) FROM FILENAME
```
("5", in this example, is the length of the character string to convert to)

Example 5: Converting JD Edwards dates to readable format using SQL
JD Edwards* ERP stores dates in a particular format:
- first digit is a 0 for dates prior to Y2K and 1 for dates after Y2K
- second and third digits are the last two numbers of a year i.e. for 2005 this value will be "05"
- third, fourth and fifth digits are the Julian day (nth day of the year)

For example, 103330 really translates to November 26, 2003. While this works well for a computer, it is not very readable by people. This can be easily changed for reading and reporting with SQL by using the following SQL formula, here is an example:
```
SELECT DATE(CHAR(1900000+SDUPMJ)) FROM F4211
```

Example 6a: Retrieving the N Largest Values of a Column – The SLOW Way
SQL methods can be mixed and combined to produce different results. This example demonstrates a method to retrieve N of the largest quantities in a data set. In this case, we will use the 5 largest customers, in terms of sales. The example below does have one draw-back – it uses a sub-select, which typically is not very efficient, so using this method on a large amount of data may cause a lot of disk access – resulting in a hit on performance.

```
SELECT * FROM CUSTOMER A
WHERE 5 > (SELECT COUNT(*) FROM CUSTOMER B
WHERE B.SALES > A.SALES )
ORDER BY A.SALES DESC
```

Example 6b: Retrieving the N Largest Values of a Column – The FAST Way
The "FETCH FIRST n ROWS ONLY" clause can help in limiting the ouptut of a query. It is also much more efficient than the query above. In this example, only 5 disk retrieval operations are necessary and if your table is indexed, you will not even need to figure out the order in which to pull the data.

```
SELECT * FROM CUSTOMER A ORDER BY A.SALES DESC FETCH FIRST 5 ROWS ONLY
```

Example 7: Finding distinct values with SQL
SQL can conveniently find distinct values in a data set, essentially, giving you only unique values. If for example, you had a mail order business and wanted to know, using the order file, what are the distinct cities from where all the orders came from, you could say:

```
SELECT DISTINCT CITY_NAME,
FROM ORDERS
ORDER BY CITY_NAME
```

Example 8: Aggregating Data with SQL

SQL can aggregate values, such as COUNT(*), SUM(COL_NAME) or AVG(COL_NAME). This method shows how one can display the resulting data by descending order of the aggregated value. The "ORDER BY 2 DESC" notation is not a typo. It means "order by the second argument in descending order".

```
SELECT CITY_NAME, SUM(ORDER_VALUE)
FROM ORDERS
GROUP BY CITY_NAME
ORDER BY 2 DESC
```

A word about NULLs
The NULL value is different from a blank value. It is a value equivalent to "unknown". Where a value is not found in a left outer join for example, a NULL is returned by SQL, essentially meaning that this value is unknown.

In SQL, one can purposely set a value to NULL by coding:
`UPDATE TABLE_A SET USER_NAME = NULL`
However, to use the existence of a NULL for comparison, one has to write
`UPDATE TABLE_A SET FIRST_NAME = '???' WHERE LAST_NAME IS NULL`

Using the DB2 equivalent of Oracle's DUAL table
In Oracle, there is a handy system table named simply "DUAL". It is used when one needs to retrieve system values, such as the system time. The DB2 for iSeries equivalent of Oracle's "DUAL" is `SYSIBM/SYSDUMMY1`. The name is longer than just "DUAL", but it works the same way. Here are two examples with dates. Note the second one, which is used to make date calculations based on today's date.

```
SELECT CHAR( DATE(TIMESTAMP('2005-12-31-23.59.59.000000'))) FROM SYSIBM/SYSDUMMY1
12/31/05

SELECT CHAR( DATE(TIMESTAMP('2005-12-31-23.59.59.000000') +7 DAYS)) FROM SYSIBM/SYSDUMMY1
01/07/06
```

In part 2 of this article, I will show you how to put it all together and use it in real (production) life.

PART 2: Running SQL on iSeries

In Part 1 of this article, I have covered a number of every-day useful examples of code that can be used for a variety of situations. While impossible to cover all possibilities, it is designed to start the reader into thinking of real-life applications SQL can be used for in a production environment. We are now at the start of Part 2, which will cover the implementation methods in which all these coding examples can be packaged into code that can run or be scheduled like any other object.

While running the odd SQL function on an interactive screen can be helpful to debug data problems, there is much more to SQL than being a better version of DSPPFM. The real power of SQL only comes when it is used within an application.

There are three ways of running SQL code on the iSeries. They are:
1) Interpreted SQL code
2) SQL code embedded in a C, RPG etc… high level language program
3) SQL Stored Procedures

I will concentrate only on the Interpreted SQL and Stored SQL procedure methods, which use SQL code ONLY. We will not cover SQL embedded in another language. This also means that we will only cover SQL code executable in batch mode.

RUNSQLSTM - Running Interpreted SQL Code in Batch
Interpreted SQL is a simple method. Write your SQL code into an ordinary source member and use the `RUNSQLSTM` command to run it. Each SQL statement must end with a semi-colon character if there is more than one statement in an individual source member. No compiling is necessary. The syntax will be checked at run time and logged in a spool file. This type of procedure cannot be debugged using a debugger, but the spooled output is typically explicit and easy to read.

There are two base points to remember when using SQL in batch on the iSeries.

1) Selects must have an output

With few exceptions, the examples shown in Part 1 are of the "SELECT" variety. A "SELECT" SQL statement all by itself works well in interactive mode because the output goes to the screen by default (it can also be configured to output to a file). When using SQL in batch, the output HAS to be directed somewhere. A SELECT in batch all by itself will produce results that will have nowhere to go and this will result in an error. The typical construct is:

```
INSERT INTO EXTRACT
SELECT INPUT.FIRST_NAME, INPUT.LAST_NAME, INPUT.SALARY
       FROM PAYROLL INPUT
       WHERE (INPUT.SALARY IS > 1000000);
```

In this case, the output of the SELECT statement is directed to the table EXTRACT

2) Set Processing Only

When using interpreted SQL on the iSeries, one can only process data in sets (as opposed to record-by-record). Record-by-record processing requires the declaration of an SQL CURSOR, and this can only be handled either in a Stored Procedure or in SQL code embedded in an ILE language such as ILE C or ILE RPG, all of which are compiled.
Once the source member containing the SQL statement is built, you can execute it in interpreted mode with the following command: RUNSQLSTM SRCFILE(SRCFILLIB/SRCFIL) SRCMBR(SRCMBR). This is a CL command and can be entered into any CL program.

SQL Stored Procedures - Running Compiled SQL Code in Batch

Creating and running SQL stored procedures is a bit more involved than interpreted SQL, but not by much. The benefit of the extra effort is worthwhile for the following reasons:
Stored Procedures

1) Compile into executable objects
    a. Compiled code runs faster than interpreted code
    b. With compiled code, one can put the program in debug and retrieve the DB2 SQL Optimizer messages to figure out the best indexes to use
2) Allow the choice of using set processing or record-by-record processing, using SQL cursors
3) Allow programming constructs such as loops and if-then-else conditions
4) Can take advantage of compiled SQL Functions

Below, is an example of a short procedure, with a do-loop and a cursor, which enables the SQL code to target data one record (row) at a time. In this case, we are updating the time stamp for each row in a table. The time stamp precision extends to the millionth of a second. Because of this level of precision, each record will be stamped with a unique time stamp value.

```
CREATE PROCEDURE PROC_NAME
LANGUAGE SQL

-- START PROCEDURE
-- This procedure will, for each row of table ER400SX, retrieve the current timestamp
-- and update the column PUBLISH_TMS within ER400SX

BEGIN

-- DECLARE CURSOR VARIABLES
DECLARE PUBLISH_TMS          TIMESTAMP ;
DECLARE WORK_TIMESTAMP       TIMESTAMP ;
DECLARE SQLSTATE             CHAR(5) DEFAULT '00000' ;
DECLARE AT_END               INT DEFAULT 0 ;
DECLARE SQLCODE              INT DEFAULT 0 ;

DECLARE CURSOR_UPD CURSOR FOR
SELECT PUBLISH_TMS FROM ER400SX MAIN;
SET AT_END = 0;

OPEN  CURSOR_UPD ;

FETCH_LOOP:

LOOP
    FETCH CURSOR_UPD INTO WORK_TIMESTAMP ;

    IF SQLCODE = 0  AND SQLSTATE = '00000' THEN
       UPDATE ER400SX
```

```
        SET PUBLISH_TMS  = CURRENT TIMESTAMP,
            TIME_ELAPSED = DAY(CURRENT_TIME_STAMP - WORK_TIMESTAMP)
        WHERE CURRENT OF CURSOR_UPD ;
    ELSE
        SET AT_END = 1 ;
        LEAVE FETCH_LOOP ;
    END IF ;

END LOOP  ;

CLOSE CURSOR_UPD ;

-- END PROCEDURE
END
```

There are several steps to turn this source code into a running, callable program.

The first step is to execute the SQL source, as one would with interpreted SQL, using the `RUNSQLSTM` command we are now familiar with.

Because the source code begins with "`CREATE PROCEDURE PROC_NAME`", running the code with RUNSQLSTM within will effectively trigger the creation of the procedure PROC_NAME. Note that if an object of type *PGM with the same name already exists in your CURRENT LIBRARY, `RUNSQLSTM` will not replace the object. The compilation will simply crash. If you want to create the procedure elsewhere, change your CURRENT LIBRARY to the desired target library prior to running the `RUNSQLSTM` command.

Assuming there are no syntax errors, an object of type CLE will be created, bearing the name of the procedure that you chose. In effect, the Operating System will wrap your SQL code in ILE C and created an ILE C object.

There is one catch with this type of program. Stored Procedures looks like ordinary ILE C programs but cannot be called from the command line. They have to be called from an SQL environment. I say this here because it is not very obvious in the manual. If called from the command line, like any other program, the Stored Procedure object will crash with a "Pointer not set for location referenced" error. The reason for this is that the SQL environment passes hidden parameters when SQL Stored Procedures are called from there.

There are four ways to call SQL procedures from an SQL environment:

1) Interactively – This is a quick way to test this type of program, provided you are using small data sets.
   a) Use the STRSQL command to enter an SQL environment.
   b) From the SQL command line, use the same syntax as you would for any program:
      `CALL LIBNAME/PROC_NAME`. This will call the Stored Procedure object and it will end normally.

2) Batch – Option 1 – This is the simplest way to write code that can be called from a program. It is a bit awkward, but it does work. Remember here that the aim is simply to call the program "from an SQL environment".
   a) You would first need to create a source member that would have inside "CALL LIBNAME/PROC_NAME"
   b) A CL program can subsequently run this member with the `RUNSQLSTM` command.

3) Batch – Option 2 – This method uses the QMQRY, or Query Manager product.
Here again, the aim is to call the program "from an SQL environment".
   a) In a QMQRY member, type `CALL LIBNAME/PROC_NAME`. Note that QMQRY offers some flexibility here, you could create a `QMQRY` designed simply to run procedures and have something like `CALL &LIBNAM/&PROC_NAME`, where `&LIBNAM` and `&PROC_NAME` could be passed as parameters.
   b) Use the `STRQMQRY` command to execute the `QMQRY` member, with or without parameters, depending on how you coded the QM source member.

4) Batch – Option 3 – This method uses a widely available command called EXCSQL, originally created by Dan Riehl (if IBM could put such a command in the OS, it would be very practical). The EXCSQL command effectively lets you execute individual SQL statements from a CL program. It can be downloaded from the Internet. Once you have built this command on your system, you can do something like
`EXCSQL REQUEST('CALL LIBNAME/PROC_NAME')`

There is one more important point on the topic of Stored Procedures. Being procedures, but also being CLE-type programs, they can be debugged. Here is the method to compile a procedure in a way that it can be debugged:

Back to the creation of the procedure using RUNSQLSTM, to be able to debug the procedure to be created, use the following option: RUNSQLSTM with DBGVIEW(*LIST) or DBGVIEW(*SOURCE). If you use *LIST, you will have a debug view of the C code generated by the system to wrap around your SQL code. If you use *SOURCE, you will only see your SQL source as you debug it. While the *LIST view is interesting to see, the *SOURCE view is much more convenient for debugging.

Once the procedure is compiled, use STRDBG PGM(PROC_NAME) UPDPROD(*YES) from the command line to go into debug mode. The next thing you will see is a screen full of source code. Hit F10 and you will be back to the command line. From there, do a STRSQLSTM and from there, a CALL PROC_NAME. You will be able to step through your program in debug mode.

Here are a few of pointers for debugging an SQL procedure:

1) In a Stored procedure, the SQLCODE is a results indicator variable affected by each database operation. A zero value in the SQLCODE indicates success. To see the value of the SQLCODE variable, use EVAL SQLCODE. To see ALL local variables at any point in your C/400 debug session, enter the following command on the debug screen command line: EVAL %LOCALVARS

2) To see the value of a character string in a C/400 debug session, use the following command:

EVAL *variablename :s 12 (12 is the length of the string you wish to display)

3). For a more in-depth understanding of what is happening with your latest SQL database operation, if there is a problem, the place to look is a system-maintained data structure named "sqlca". To be able to see the contents of this data structure, in a debug session, use **EVAL sqlca** (Make sure that "sqlca" is in lower case). This instruction will display the entire SQLCA data structure. Remember, these are C programs and all variables are case-sensitive.

For more information, look up IBM Red Book "Stored Procedures, Triggers and User Defined Functions on DB2 Universal Database for iSeries". Further definitions of the sqlca data structure are available on IBM's website in the DB2 Application Development Guide.

SQL Functions – Recycle that Code!
One of the features I particularly like about SQL is functions. In the same way that on the iSeries, you can create your own commands, you can create your own SQL functions and use them just like any other SQL system-supplied function. While procedures can receive and return many parameters, functions can receive many but will only return a single value. The way functions are implemented on the iSeries is as follows.

To compile a function, use the RUNSQLSTM command. This will compile the function into an object of type *SRVPGM. This means that the function cannot be called on its own but it can be evoked from within an ILE program or an SQL Stored procedure. Below, is an example using dates to grade an item as fresh, old or very old:

```
CREATE FUNCTION HOW_OLD (INDATE DATE)
  RETURNS CHAR(8)
  LANGUAGE SQL
BEGIN
   DECLARE  HOW_OLD CHAR(8);
   DECLARE  RETVAL CHAR(8);
    CASE
       WHEN INDATE < CURRENT_DATE - 60 DAYS THEN
       SET RETVAL = 'VERY OLD';
       WHEN INDATE < CURRENT_DATE - 30 DAYS THEN
       SET RETVAL = 'OLD';
    ELSE
       SET RETVAL = 'FRESH';
    END CASE;
   RETURN(RETVAL);
END
```

Once built, the HOW_OLD function can be re-used in any number of stored procedures, or even other functions.
In a stored procedure, using the HOW_OLD function would be possible with the following syntax:

```
AGE_GRADING = HOW_OLD(DATEVAR)
```

Essentially, the HOW_OLD function has become a black-box, nobody needs to know how it does its work, as long as one knows what it needs and what it will return. Another practical function that can be put to use immediately if your company

runs JD Edwards software is a Julian Date conversion function, which would receive the Julian date and return a regular format date (see example 5 in Part 1 of this article).

In Real Life
Using SQL day-in and day-out, one quickly realizes that SQL is one of these languages where it is deceptively easy to code programs which work well with a small data samples but do not scale well and sit for hours on large data sets. The usual reason for this behavior is either lack of proper indexes or a choice of SQL method that is not optimum.

There are some basic rules to avoid this type of situation:

1) Indexes, Indexes, Indexes. For best performance, it is well worth to ensure indexes are present with the keys you will use in your joins. If possible, create your indexes with the keys of lowest cardinality first and highest cardinality last. This will also speed up the processing. Code your SQL join statements with keys that match the order of the indexes you created. Consider building new indexes if none of the existing ones fit your current requirement. The SQL Optimizer, part of the operating system, will look for the best combination of indexes for you.

2) Use proper data types: If at all possible, store data in such a way that you will not need to manipulate it – for example, use numeric types for numbers, date or timestamps types for dates. SQL has powerful casting functions but as mentioned previously, when you start using function results as part of a join, existing indexes cannot be used.

3) Don't be afraid to use work files. There are times when you can achieve better performance by using several simple SQL instructions with intermediate work tables than by trying to cram everything into one long and overly sophisticated command.

4) Maintainability and performance: These are goals to shoot for. Example 6, in Part 1 of this article demonstrated a method to retrieve N of the largest quantities in a data set with a single SQL statement. This SQL statement, while elegant, accomplished its goal with a high IO cost. No problem on a small table, but on a large set, it is an expensive statement. This is an example where a cursor could be more efficient, having the ability to retrieve the N biggest amounts from a table with exactly N accesses to the database using a descending order index to read the table – no matter how big the table is. With experience, you will become better at figuring the most efficient SQL method for a given situation. This is one where you learn every day. The more you use it, the more you learn.

5) Interpreted vs. Compiled: The `RUNSQLSTM` interprets the SQL commands written in a source member. This is the fastest way to implement data set processing SQL code. Stored procedures are compiled, which means a few extra steps to implement and the necessity to manage objects as well as source. Stored procedures allow both data set processing and row-by-row processing using cursors. They typically run faster than interpreted code, and also allow one to figure out what the SQL Optimizer needs to do its job most efficiently. For large volume applications, where processing speed is critical, you may want to consider stored procedures.

Method to find the SQL Optimizer suggestions to improve program or stored procedure SQL performance
The DB2 Optimizer is part of the iSeries OS. The purpose of this operating system component is to ensure any SQL statement submitted on the system will use the most efficient access path available to retrieve the data. If you find your SQL stored procedure does not run as fast as you expected it to, with the method explained below, you can find out what indexes the DB2 optimizer is looking for in terms of indexes. This method is also valid for SQL embedded in C or RPG.

1) Go in debug and change the job to record all activities and second level text
```
STRDBG UPDPROD(*YES)
CHGJOB LOG(4 4 *SECLVL)
```
Note: with `*SECLVL`, both the message text and the message help (cause and recovery) of the error message are written to the job log.

2) Call the Stored Procedure from within an SQL environment (`STRSQL` and do "`CALL LIBNAM/PROC_NAME`" from the SQL command line or `RUNSQLSTM` of a member containing "`CALL LIBNAM/PROC_NAME`")

3) `SIGNOFF LOG(*LIST)` (I use this method, as opposed to looking at the joblog because it is easier to scan a spool file)

4) Signon again and look for your last job log with `WRKSPLF` (the name of the file will be `QPJOBLOG`)
In that spool file, find the following character strings:
`**** Starting optimizer debug message for query`
and `Access path suggestion for file` (Note: DB2 Optimizer suggestions may or may not appear)
They will show you what the SQL optimizer is looking for and the suggestions it may have made for you to improve the

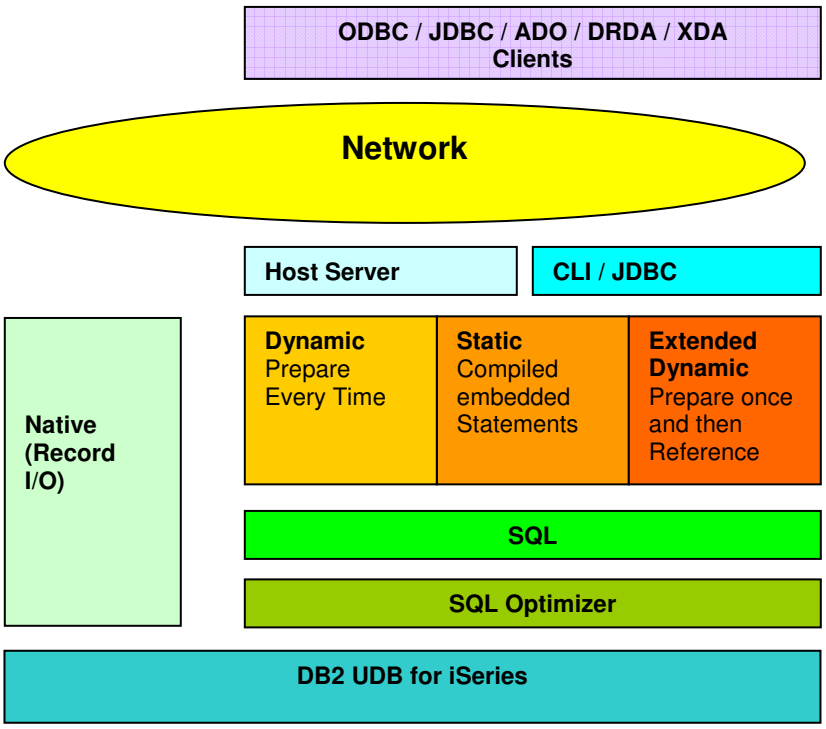
performance of your SQL code. Typically the Optimizer will suggest indexes, with precise key orders.

Once you know what the Optimizer wants, you build the suggested indexes and re-run your job. Nine times out of ten, you should see a significant performance improvement in your SQL job. Note that this method is also valid for SQL embedded in C or other languages.

Another tool that can yield useful information is the `PRTSQLINF` command. It prints the query implementation plan contents for any high level language program, stored procedure or service program with embedded static SQL. `PRTSQLINF` produces a spooled file containing the query optimizer index choices for the SQL statements contained in the program or procedure. Knowing what indexes the optimizer is planning to use may save you some debugging time. Keep in mind that PRTSQLINF does not make index suggestions.

Implementation Considerations
One of the critical considerations if you are going to start using SQL for production programs is your promotion control software. Basically, if you cannot promote it, you cannot use it very conveniently or on a larger scale. Promoting SQL procedures to a production environment means different steps from a traditional compilation have to be executed. They are not very complicated but they are different. If you are aiming to use SQL in production jobs, do look at this particular aspect ahead of time. Make sure your implementation software can handle SQL code.

In Conclusion: What's Next?
SQL is a vast topic. In this article, we only scratched the surface. The diagram below is copied from an IBM presentation titled "Tuning Web-based Data Access for DB2 UDB for iSeries".



Take note of the contrast between the left side (the Native Record I/O) vs. the right side (the SQL-based access) of the diagram. Clearly, IBM is investing in SQL-based connectivity, making the iSeries database compatible with - and accessible via all the leading data exchange protocols. To remain competitive in the now highly commoditized server market, the iSeries, long considered a niche machine, now supports the following client protocols:

- **ODBC** - Open Data Base Connectivity, a standard database access method developed by the SQL Access Group to make it possible to access any data from any application, regardless of which database management system (DBMS) is handling the data.
- **CLI** - Call Level Interface (DB2 CLI) is IBM's callable SQL interface to the DB2 family of database servers. It is a 'C' and 'C++' application programming interface for relational database access that uses function calls to pass dynamic SQL statements as function arguments.
- **JDBC** , Java Database Connectivity, a Java API that enables Java programs to execute SQL statements. This allows Java programs to interact with any SQL-compliant database.
- **ADO**, ActiveX Data Objects, Microsoft's high-level interface for data objects.

- **DRDA**, Distributed Relational Database Architecture, an IBM standard for accessing database information across IBM platforms that follows SQL standards.
- **XDA**, Extensible Distributed Architecture, an enterprise architecture for distributed, high-performance, object-oriented, applications.

All arrows point to SQL as a base layer for modern iSeries database development and connectivity. This is now a base necessity to be competitive as a programmer. Are you fluent with SQL yet? Right now, you can be. On your iSeries, you've got the power.

The author acknowledges the proof-readers who contributed generously to this article.