Technical Report

Santa Teresa
Laboratory
San Jose, CA

IBM

TR

INTERACTIVE SQL AND APL2

by Nancy Wheeler

May 1986

TR 03.289

TR 03.289

INTERACTIVE SQL AND APL2

May, 1986

Nancy Wheeler

IBM
General Products Division
Santa Teresa Laboratory
San Jose, California

ABSTRACT

This paper is a tutorial on the use of AP 127, an APL2 auxiliary processor which interfaces to the relational databases SQL/Data System (SQL/DS) and IBM Database 2 (DB2) from the APL2 workspace.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

INTRODUCTION

AP 127 allows APL2 programmers to imbed Structured Query Language (SQL) statements in their APL2 functions, just as SQL statements can be imbedded in COBOL or ASSEMBLER programs. Unlike those languages, however, APL2 allows interactive access to the databases. Also, because APL2 does not have explicit data declarations, SQL declarations and control blocks do not need to be coded in APL2. The formalities of connecting to the database and passing it requests in the proper format are taken care of by AP 127; the user need only construct the SQL statements.

The AP 127 interface to SQL is simple and takes advantage of APL2 arrays. One shared variable is used for return codes and data, and the SQL tables are returned to the APL2 workspace as nested arrays. Once the data is in the workspace, the full power of the APL2 language is available for manipulating it.

Along with AP 127, a workspace called SQL is distributed with the APL2 product. The SQL workspace contains APL2 functions for using AP 127. In general, when functions are referred to in this paper, but their source code is not shown, they are included in the SQL workspace. Functions whose code is shown are not included in SQL.

NOTE: The interface protocols are generally the same when interfacing to IBM Database 2 (DB2) and SQL/Data System (SQL/DS). Unless specifically stated, all references to SQL in this document apply to both environments.

MIXED, NESTED DATA

Figure 1 shows a nested array that is typical of what an SQL table might look like. The examples used in this presentation are based on a fictional company, ABC Limited, and the data here is contained in one APL2 variable, ABCT.

```
      ABCT
ADAMS    12 3 36000 ADMIN PA1
BANKS    15 3 35000 SALES PS2
CROW      6 2 24000 PROD  PP1
DEAN     12 3 38000 PROD  PP2
EATON    18 4 40000 RES   PR1
FARR     25 5 50000 ADMIN PA1
GALVIN    5 3 27000 SALES PS1
HARVEY   23 5 45000 SALES PS1
INGRAM    2 1 18000 ADMIN PA2
JACKSON   1 1 16000 PROD
KAHAN     6 3 32000 RES   PR2
LAMAR    21 5 45000 PROD  PP1
MULVEY    3 2 21000 SALES PS2

      DISPLAY 4↑[1]ABCT
```

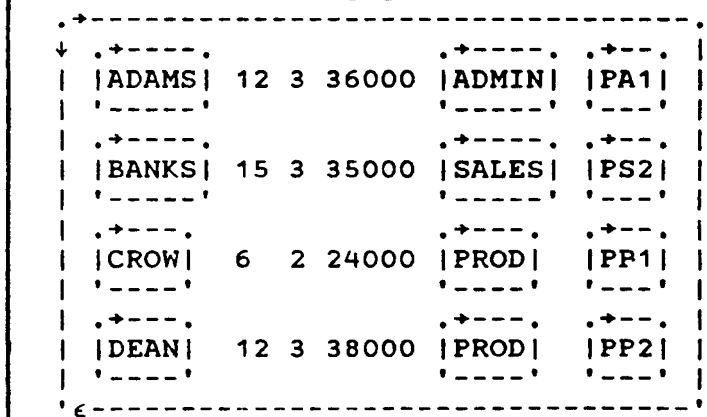

Figure 1. Variable with relational-like data

The DISPLAY workspace, included with APL2, allows a pictorial representation of data, making it easier to see the type and structure of an object. Here, we display the first four rows of the ABCT variable for demonstration. Note that the numbers are scalars, so they have no boxes around them. The character items are vectors, and the entire object is a matrix.

COLUMN HEADINGS

In a relational database we use the concept of data in tables with rows and columns, and the columns have names associated with them. In Figure 2 we assign to the APL2 variable ABCH a vector of character items representing the names of the columns of our data.

Use of the DISPLAY function shows that ABCH is a vector of character vectors.

```
      ABCH
 NAME YOS LEVEL SALARY DEPT PROJ

      DISPLAY ABCH
.→-----------------------------------------------.
| .→---. .→--. .→----. .→-----. .→---. .→---. |
| |NAME| |YOS| |LEVEL| |SALARY| |DEPT| |PROJ| |
| '----' '---' '-----' '------' '----' '----' |
'∈-----------------------------------------------'
```

Figure 2. Column names in APL2 format

PUTTING IT TOGETHER

Given that we now have some data in one variable, and some column headings in another, we might want to combine these variables into a format suitable for a report. The APL2 function PRESFORM will do that. Line 2 of the function places a row of "=" signs under the headings to delimit them from the data, and line 3 concatenates the data onto them.

```
      ∇PRESFORM[⎕]∇
    ∇
[0]   Z←H PRESFORM T
[1]  ⍝ PRESENTATION FORMAT
[2]   Z←⊃H((⍴⍕H)⍴⍕'=')
[3]   Z←Z,[⎕IO]T
    ∇ 1984-05-31 16.32.57 (GMT-8)
```

Figure 3. PRESFORM function

To invoke the PRESFORM function, we use the headings (ABCH) as the left argument and the data (ABCT) as the right argument.

```
      ABCH PRESFORM ABCT
 NAME     YOS LEVEL SALARY DEPT  PROJ
 ====     === ===== ====== ====  ====
 ADAMS     12     3  36000 ADMIN PA1
 BANKS     15     3  35000 SALES PS2
 CROW       6     2  24000 PROD  PP1
 DEAN      12     3  38000 PROD  PP2
 EATON     18     4  40000 RES   PR1
 FARR      25     5  50000 ADMIN PA1
 GALVIN     5     3  27000 SALES PS1
 HARVEY    23     5  45000 SALES PS1
 INGRAM     2     1  18000 ADMIN PA2
 JACKSON    1     1  16000 PROD
 KAHAN      6     3  32000 RES   PR2
 LAMAR     21     5  45000 PROD  PP1
 MULVEY     3     2  21000 SALES PS2
```

Figure 4. Invocation of PRESFORM

USING SQL FROM APL2

Now that we have seen what we might do with SQL-like data in the APL2 workspace, we need to know how to create and retrieve actual SQL data using APL2.

TABLE CREATION

The first step is the creation of the table. To do this we use an ordinary SQL CREATE statement; we have placed the statement in a character variable called ABCC.

In addition to the CREATE, we will need an SQL INSERT statement to insert data into the table. The variable ABCI contains the INSERT.

```
      ABCC
CREATE TABLE ABC
(NAME VARCHAR(20),
 YOS  SMALLINT,
 LEVEL SMALLINT,
 SALARY INTEGER,
 DEPT VARCHAR(8),
 PROJ CHAR(3))

      ABCI
INSERT INTO ABC
 VALUES(:1,:2,:3,:4,:5,:6)
```

Figure 5. SQL statements in APL2 variables

In programming languages such as PL/I and COBOL, when you want to execute a statement a number of times, you use variable names preceded by colons to indicate that the data will be found later in those variables. In APL2, we use numbers preceded by colons. The numbers represent indices into an APL2 vector. AP 127 will remember the indices and replace them with the DYNAMIC SQL placeholder "?" before passing the statement to SQL. When we execute the statement, AP 127 will use the indices to get the data from the APL2 vector, and pass the values to SQL.

The table is created in three steps using the APL2 SQL function, which is included in the SQL workspace distributed with APL2. First we execute the CREATE statement, and then the INSERT statement. We pass the data, which is already contained in the variable ABCT, as the second parameter to the SQL function on the INSERT. Since ABCT is a matrix of data, the SQL function will issue the INSERT statement once for each row of the matrix. We call the data matrix (or vector) a "value-list".

Finally, we issue a COMMIT to make the additions permanent in the database. If we did not want the changes to be permanent, we could have issued a ROLLBACK instead.

```
      SQL ABCC
0 0 0 0 0

      SQL ABCI ABCT
0 0 0 0 0

      COMMIT
0 0 0 0 0
```

Figure 6. Creating a table

NOTE: AP 127 does not do any implicit COMMITs. If no COMMIT is done, work will be rolled back upon retraction of the variable shared with AP 127. This is consistent with the workspace)SAVE conventions in APL2.

The UNTIL function in the SQL workspace allows us to execute a sequence of SQL commands. It takes a vector of commands and executes them until a non-zero return code is encountered, or until the commands are exhausted. This is valuable, since in most cases we will not want to do a COMMIT if an error has occurred during the execution of one of the commands. Figure 7 shows an alternative way to execute the table creation sequence.

```
      SQL UNTIL ABCC (ABCI ABCT) 'COMMIT'
  0 0 0 0 0       0 0 0 0 0       0 0 0 0 0
```

Figure 7. The UNTIL function

TABLE RETRIEVAL

To retrieve an SQL table, we can also use the SQL function. We will pass an SQL SELECT statement to the function and assign the result to a variable called RESULT.

```
      RESULT←SQL 'SELECT * FROM ABC'

      1⊃RESULT
0 0 0 0 0
      2⊃RESULT
ADAMS    12 3 36000 ADMIN PA1
BANKS    15 3 35000 SALES PS2
CROW      6 2 24000 PROD  PP1
DEAN     12 3 38000 PROD  PP2
EATON    18 4 40000 RES   PR1
FARR     25 5 50000 ADMIN PA1
GALVIN    5 3 27000 SALES PS1
HARVEY   23 5 45000 SALES PS1
INGRAM    2 1 18000 ADMIN PA2
JACKSON   1 1 16000 PROD
KAHAN     6 3 32000 RES   PR2
LAMAR    21 5 45000 PROD  PP1
MULVEY    3 2 21000 SALES PS2
```

Figure 8. Issuing a SELECT

The first item in the result variable is the return code vector. If all zeros, the statement was successfully processed. The second item is the result data; the data returned is in the same array format as the data we used to create the table. (After every AP 127 operation, a two-item vector is returned. For operations that return no data, such as CREATE and INSERT, the second item is null.)

NOTE: The table retrieved does not have to have been created with APL2. Any table in the database that the user has authority to access may be retrieved.

COLUMN NAME RETRIEVAL

Now that we have retrieved the table, the next step is to retrieve the column headings. This is done with the AP 127

DESCRIBE operation, which is analogous to an SQL DESCRIBE, but does not cause an SQL DESCRIBE. The AP 127 DESCRIBE returns to the user the column information obtained when the SQL DESCRIBE was done. The SQL workspace function DESC is used to execute the operation.

```
      DRESULT←DESC 'APL2'

      1⊃DRESULT
0 0 0 0 0

      2⊃DRESULT
NAME  YOS  LEVEL  SALARY  DEPT  PROJ
V 20  S    S      I       V 8   C 3

      (2⊃DRESULT)[1;]
NAME  YOS  LEVEL  SALARY  DEPT  PROJ
```

Figure 9. Issuing a DESCRIBE

The parameter passed to DESC is the character string 'APL2'. APL2 is the statement name used by the SQL function to prepare SQL statements. Since we used the SQL function to issue our SELECT, we use that same name to issue the DESCRIBE.

The result of the DESCRIBE is, as usual, a two-item vector. The first is the return code vector, and the second the data. In the case of DESCRIBE, the data consists of the names and data types of each of the columns. To isolate the names, we index the first axis of that data.

Figure 10 shows a short APL2 function which will select the column headings. As we have done, it uses the DESC function to get the DESCRIBE information, and then indexes the first axis.

```
       ∇HEADS[□]∇
    ∇
[0]    Z←HEADS NAME
[1]   ⍝RETURNS COLUMN HEADINGS
[2]    Z←DESC NAME              ⍝ SPEC,REF
[3]    Z←((1+□IO)⊃Z)[□IO;]           ⍝ GET TITLES
    ∇ 1986-03-23 15.19.09 (GMT-8)
```

Figure 10. HEADS function

A FINISHING TOUCH

The TABLE function combines all the steps of table retrieval. First it issues the query using the SQL function. Then it calls the HEADS function to get the column names. Finally, we put it all together using the PRESFORM function defined earlier.

```
      ∇TABLE[□]∇
    ∇
[0]   Z←TABLE STMT;T;H
[1]  ⍝ BUILD A REPORT-FORM TABLE
[2]   T←2⊃SQL STMT      ⍝ GET THE TABLE
[3]   H←HEADS 'APL2'    ⍝ GET THE HEADINGS
[4]   Z←H PRESFORM T    ⍝ BUILD THE REPORT
    ∇ 1986-03-23 15.20.27 (GMT-8)
```

Figure 11. TABLE function

The table function accepts the SQL SELECT as its parameter, and returns the finished result.

```
      TABLE 'SELECT * FROM ABC'
NAME     YOS LEVEL SALARY DEPT  PROJ
====     === ===== ====== ====  ====
ADAMS     12     3  36000 ADMIN PA1
BANKS     15     3  35000 SALES PS2
CROW       6     2  24000 PROD  PP1
DEAN      12     3  38000 PROD  PP2
EATON     18     4  40000 RES   PR1
FARR      25     5  50000 ADMIN PA1
GALVIN     5     3  27000 SALES PS1
HARVEY    23     5  45000 SALES PS1
INGRAM     2     1  18000 ADMIN PA2
JACKSON    1     1  16000 PROD
KAHAN      6     3  32000 RES   PR2
LAMAR     21     5  45000 PROD  PP1
MULVEY     3     2  21000 SALES PS2
```

Figure 12. Execution of TABLE

We should point out here that the functions HEADS and TABLE are purposely made very simple to demonstrate the basic idea. In a real application, however, you would want to add error-checking, and probably make them more sophisticated.

The QUERY function in the SQL workspace does that, and QUERY calls other functions that are user- modifiable to add the headings, combine result tables, and manipulate the result data in a customized way.

GETTING FANCIER

DATA INPUT FUNCTIONS

Figure 13 demonstrates the use of the IN function, also in the SQL workspace. Assign IN to a variable (here SALESQ) and IN will prompt you line-by-line for your SQL statement, building a character matrix for you. When you are done, you enter a null line to complete the matrix. This allows easy entry of long queries.

```
     SALESQ←IN
SELECT NAME,SALARY
FROM ABC WHERE
DEPT = 'SALES'

      SALESQ
SELECT NAME,SALARY
FROM ABC WHERE
DEPT = 'SALES'
```

Figure 13. The IN function

We can use our TABLE function again to execute the SALESQ query, which is now contained in an APL2 character variable.

```
      TABLE SALESQ
 NAME   SALARY
 ====   ======
 BANKS   35000
 GALVIN  27000
 HARVEY  45000
 MULVEY  21000
```

Figure 14. Execution of a pre-defined query

Two other functions in the SQL workspace make data input easier. EVAL and EVALSIM take data from an APL2 character array and create a nested array of the correct format to pass to SQL. EVAL does this based on the data in the array,

and EVALSIM does it based on descriptions of the data. These functions make it possible to enter data using the IN function, or perhaps a system editor, rather than creating the nested matrix directly.

GENERALIZING A QUERY

If we want to make the above SQL statement more general, we can substitute for the 'SALES' department name a host variable index. We will place this more general query in the DEPTQ variable.

```
      DEPTQ
SELECT NAME,SALARY
  FROM ABC WHERE
  DEPT = :1
```

Figure 15. Query with host variable indices

In the next section we will execute DEPTQ with AP 127.

USING AP 127 OPERATIONS

When the SQL function is used to execute a query, it generates a stack of the proper sequence of AP 127 commands necessary for that execution. The SQL function is easy to use, and does note require detailed knowledge of the AP 127 interface. We could execute DEPTQ using the SQL function.

It is also possible to bypass the use of the SQL function and pass the commands to AP 127, either directly using shared variable operations or with the cover functions provided in the SQL workspace. There is a cover function for each AP 127 operation. We have used one already, the DESC function. Now we will use some others to execute our generalized query.

Using AP 127 operations requires a little more knowledge about how SQL queries are actually executed. It happens in several steps:

1. First, we issue the PREP command to PREPARE the statement. The PREPARE causes SQL to parse the SQL statement. We must name our statement so that SQL can distinguish it from other statements we PREPARE. We will call it 'ABC'.

2. Next, we issue the OPEN command. If the statement we prepared had any host variable indices in it, we also pass the vector of items (value-list) to be substituted for them. Here, we have only one host variable index, for the department name. Since the character vector we are passing is to be considered as only one item by AP 127, we must enclose it. If we did not, each letter in the vector would be considered a separate item.

3. To retrieve the table, the FETCH command is used, and the data can then be assigned to a variable to retain it for processing.

We can at this point go back and repeat the OPEN and FETCH steps as many times as desired, using a different department name each time.

4. Finally, we issue a CLOSE command to close the cursor.

Figure 16 shows the sequence used to execute SALESQ.

```
      PREP 'ABC' DEPTQ
0 0 0 0 0

      OPEN 'ABC' (⊂'SALES')
0 0 0 0 0

      SALES←2⊃FETCH 'ABC'

      OPEN 'ABC' (⊂'ADMIN')
0 0 0 0 0

      ADMIN←2⊃FETCH 'ABC'

      CLOSE 'ABC'
0 0 0 0 0
```

Figure 16. Using AP 127 operations

The choice of whether to use the SQL function or the AP 127 operations should be made, of course, according to the application. Using the function requires less knowledge of

SQL execution protocols. However, SQL also does some statement parsing and error-checking, each of which adds to the execution time, and may cause you to re-issue PREPARE statements unnecessarily, depending on the format of the value-list passed. In general, for ad hoc queries and prototyping, SQL is easy and quick to code. For production applications, AP 127 operations provide more control and better performance.

USE OF EACH FOR OUTPUT

Remember that once we have fetched our result tables into the APL2 workspace, we have the full power of APL2 to analyze and format the data.

This APL2 example uses the EACH (¨) operator to call the PRESFORM function for each of the variables ADMIN and SALES. The left argument is enclosed so it will be replicated and used for each call of PRESFORM. The output from the statement is a two-item vector of result tables, which looks like a very good beginning for a report to the management of ABC Limited.

```
      (⊂HEADS 'ABC') PRESFORM¨ ADMIN SALES

  NAME    SALARY      NAME    SALARY
  ====    ======      ====    ======
  ADAMS    36000      BANKS    35000
  FARR     50000      GALVIN   27000
  INGRAM   18000      HARVEY   45000
                      MULVEY   21000
```

Figure 17. Using EACH with SQL results

For those who like to use the GDDM Interactive Chart Utility to create reports, the CHART and CHARTDATA functions in the SQL workspace provide a direct link to that facility.

AP 127 UTILITY OPERATIONS

In addition to the operations which issue calls to SQL, there are some AP 127 operations to query the status of the SQL statements.

NAMES COMMAND

The NAMES command returns a list of the statement names currently active.

```
      NAMES
0 0 0 0 0   APL2 ABC
```

Figure 18. Active SQL statements

STMT COMMAND

The STMT command returns the text of the SQL statement named.

```
      2 1⍴STMT'APL2' 'ABC'
0 0 0 0 0  SELECT * FROM ABC
0 0 0 0 0  SELECT NAME,SALARY
             FROM ABC WHERE
             DEPT = :1
```

Figure 19. Text of SQL statements

We have again used EACH here to execute the STMT command on each of the names. We have then reshaped the result from a two-item vector to a matrix so that it can be displayed on our screen.

STATE COMMAND

The STATE command returns the type and status of the given statement.

The first number can be:

- 0 - Undetermined
- 1 - Non-Cursor
- 2 - Cursor (SELECT)

The second number can be:

- 0 - Unprepared (Error during PREPARE)
- 1 - Prepared
- 2 - Open (Cursor only)

```
      2 1⍴STATE¨'APL2' 'ABC'
0 0 0 0 0   2 1   CURSOR PREPARED
0 0 0 0 0   2 1   CURSOR PREPARED
```

Figure 20. State of SQL statements

When retrieving a table, AP 127 allows several options to be set to determine the format of the result table.

The options will be demonstrated using the query contained in the variable OPTSQ.

First, we execute the query using the default option settings.

```
      OPTSQ
SELECT NAME,DEPT,PROJ
FROM ABC WHERE
LEVEL < 3

     ORESULT←SQL OPTSQ

     1⊃ORESULT
0 0 0 0 0

     DISPLAY 2⊃ORESULT
```

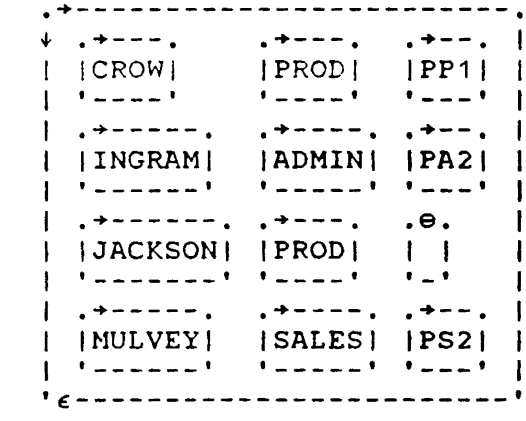

Figure 21. Default result structure

When the result table is displayed, notice that the data is a matrix, with each field of the table a distinct item in the matrix. Any null fields are represented as APL2 nulls (empty vectors). This format is known as the MATRIX option.

The GETOPT command is used to query the current option settings.

```
     GETOPT
0 0 0 0 0   MATRIX NOLENGTH  20
```

Figure 22. GETOPT operation

VECTOR OPTION SETTING

Now, we will change the first option, 'MATRIX', to 'VECTOR' and issue the same query.

```
      SETOPT 'VECTOR'
 0 0 0 0 0

      VRESULT←SQL OPTSQ

      1⊃VRESULT
 0 0 0 0 0

       DISPLAY 2⊃VRESULT
 .→-------------------------.
 | .→------. .→----. .→--. |
 | ↓CROW    | ↓PROD | ↓PP1| |
 | |INGRAM  | |ADMIN| |PA2| |
 | |JACKSON| |PROD | |   | |
 | |MULVEY  | |SALES| |PS2| |
 | '--------' '-----' '---' |
 '←-------------------------'
```

Figure 23. VECTOR format

The VECTOR option returns the table as a vector, with one item for each COLUMN of the result table. The columns are made into uniform matrices by padding the variable character items to the length of the longest one, and by replacing nulls with blanks and zeroes.

LENGTH OPTION SETTING

The vector format saves space, but some information about the actual size of the data items is lost through padding and null replacement. The LENGTH MATRIX is a way to preserve that information. We will now set the LENGTH option and reissue the query.

```
      SETOPT 'LENGTH'
0 0 0 0 0

      LRESULT←SQL OPTSQ

      1⊃LRESULT
0 0 0 0 0

       DISPLAY 2⊃LRESULT
┌→─────────────────────────────────┐
│ ┌→──────┐ ┌→────┐ ┌→──┐ ┌→────┐ │
│ ↓CROW   │ ↓PROD │ ↓PP1│ ↓4 4 3│ │
│ │INGRAM │ │ADMIN│ │PA2│ │6 5 3│ │
│ │JACKSON│ │PROD │ │   │ │7 4 0│ │
│ │MULVEY │ │SALES│ │PS2│ │6 5 3│ │
│ '───────' '─────' '───' '~────' │
'←─────────────────────────────────'
```

Figure 24. LENGTH option

The length matrix is returned as an additional item in the result vector, and gives the length of each item before padding. The length of null items is 0.

FETCH BLOCKING

SQL allows application programs to FETCH only one row of data at a time. The third AP 127 option allows the APL2 user to specify how many rows of the result table to return on each FETCH request. AP 127 will then FETCH the specified number of rows before returning to the workspace. Judicious use of the AP 127 FETCH blocking can save space and execution time, since each call to AP 127 can be constructed to return an optimum number of rows for the particular situation.

To demonstrate the use of AP 127 FETCH blocking, we will reset the options, setting the third option, the ROWS option, to a smaller number for use with our example. (The default setting is 20.)

```
      SETOPT 'MATRIX' 'NOLENGTH' 5
0 0 0 0 0

      ROWS←SQL 'SELECT NAME,DEPT,PROJ FROM ABC'

      1⊃ROWS
0 0 1 0 0
      2⊃ROWS
ADAMS ADMIN PA1
BANKS SALES PS2
CROW  PROD  PP1
DEAN  PROD  PP2
EATON RES   PR1
      3⊃ROWS
 FETCH APL2
 CLOSE APL2
```

Figure 25. ROWS option

Note that the return code from this request is not all zeroes. The (0 0 1 0 0) return code vector signals that the entire result table has not yet been fetched, and there is a third item in the result. This third item is returned only when using the SQL function, and it contains the stack of commands that were left unexecuted when the non-zero return code occurred.

The technique we will demonstrate for handling the incomplete query involved the use of another SQL workspace function, RESUME. First, the portion of the table returned is saved into a variable called TAB. The RESUME function is called with the unexecuted stack as its parameter, and execution of the query continues. The next five rows of the table are fetched.

```
      TAB←2⊃ROWS

      ROWS←RESUME 3⊃ROWS

      1⊃ROWS
0 0 1 0 0
      2⊃ROWS
FARR    ADMIN PA1
GALVIN  SALES PS1
HARVEY  SALES PS1
INGRAM  ADMIN PA2
JACKSON PROD
      3⊃ROWS
 FETCH APL2
 CLOSE APL2
```

Figure 26. RESUME execution

Our return code vector of (0 0 1 0 0) indicates that we still do not have all of the table. This result is similar to the result on the last pass, except the data is different.

To save this piece of the table, we will catenate it to the previously saved piece and assign the result back to the TAB variable.

```
      TAB←TAB,[1]2⊃ROWS
      TAB
 ADAMS   ADMIN PA1
 BANKS   SALES PS2
 CROW    PROD  PP1
 DEAN    PROD  PP2
 EATON   RES   PR1
 FARR    ADMIN PA1
 GALVIN  SALES PS1
 HARVEY  SALES PS1
 INGRAM  ADMIN PA2
 JACKSON PROD
```

Figure 27. Catenating the results

Again, we call the RESUME function, and this time our return code indicates we have fetched the entire table.

When the return code vector is (0 0 0 0 0), the third item of the result is null.

```
      ROWS←RESUME 3⊃ROWS

      1⊃ROWS
0 0 0 0 0
      2⊃ROWS
KAHAN  RES   PR2
LAMAR  PROD  PP1
MULVEY SALES PS2
      3⊃ROWS
```

Figure 28. RESUME execution again

Finally, we catenate the last piece of the table to the rest, and we have accumulated the entire result table.

```
      TAB←TAB,[1]2⊃ROWS
      TAB
 ADAMS   ADMIN PA1
 BANKS   SALES PS2
 CROW    PROD  PP1
 DEAN    PROD  PP2
 EATON   RES   PR1
 FARR    ADMIN PA1
 GALVIN  SALES PS1
 HARVEY  SALES PS1
 INGRAM  ADMIN PA2
 JACKSON PROD
 KAHAN   RES   PR2
 LAMAR   PROD  PP1
 MULVEY  SALES PS2
```

Figure 29. The final result

In the case of such a small table, of course, one would not normally fetch the result in so many steps. The technique shown, however, may be used when space considerations prohibit fetching the entire table in one pass, or when the number of rows in the result cannot be predicted.

The QUERY function, mentioned earlier, incorporates techniques like these for accumulating the result table.

AP 127 ERROR HANDLING

When an error occurs during AP 127 execution, the first item in the return code vector is 1, and the fourth and fifth items indicate the source and type of error.

The MESSAGE function may be used to retrieve the text of the message. It accepts as a parameter the return code vector from the failed operation. The result from execution of the MESSAGE function depends on the type of error and the environment.

ERRORS FOUND BY AP 127

Sometimes an error is detected in AP 127 before the request is passed to SQL. When an error is discovered by AP 127, the fourth item in the return code vector is 1, and the fifth is the error message number. In Figure 30, we have typed an incorrect option setting. MESSAGE returns the actual text of the AP 127 message that corresponds to the return code.

```
      SETOPT 'VECTOE'
 1 0 0 1 127

      MESSAGE 1 0 0 1 127
 ERROR MESSAGE.
 VECTOE IS AN UNKNOWN OPTION VALUE
```

Figure 30. An AP 127 error

SQL ERRORS

Sometimes, although AP 127 detects no error, SQL returns an error to AP 127. To show what happens in this case, we will formulate a query that will fail, and execute it using the correct AP 127 procedure. AP 127 will not discover this error, and the query will be passed on to the database.

```
      BADR←SQL 'SELECT * FROM GARBAGE'

      1⊃BADR
1 0 0 2 ¯204
      2⊃BADR

      3⊃BADR
 PREP APL2 SELECT * FROM GARBAGE
 OPEN APL2
 FETCH APL2
 CLOSE APL2
```

Figure 31. An SQL error

When an error is discovered by SQL, the fourth return code is 2, and the fifth return code is the SQLCODE.

As with the incomplete fetch case, the third item of the result contains the stack of unexecuted commands. This will tell us that the error occurred on the PREP step.

The MESSAGE function can also be used for SQL errors. It will return the contents of the SQLCA control block, and this may then be used in conjunction with the database message manual to debug the problem.

```
     MESSAGE 1⊃BADR
 ERROR MESSAGE.

  ¯204
  WHEELS   GARBAGE
  ARIXOCA
  110 0 0 ¯1 0 0
```

Figure 32. MESSAGE with an SQL error.

NOTE: When executing in DB2, the MESSAGE function will also return the text of the error message as formatted by DB2. In SQL/DS, message and help text is available in SQL tables installed with the system.

THE CONNECT COMMAND

The CONNECT command is available in SQL/DS only. This command allows you to specify the User ID that will be used in making the database connection. The facility is useful, for example, when you want only one User ID to have certain authority or access to certain tables. Then, when necessary, other users can "become" that User ID, if they know the password associated with it.

Use of CONNECT can also save keystrokes. When connected as another user, that user's ID is automatically prefixed to all table names instead of your own ID. They then do not have to be explicitly typed. Figure 33 shows an example of the use of CONNECT.

```
      SQL 'SELECT * FROM SQLDBA.INVENTORY'
 0 0 0 0 0    207 GEAR     75
              209 CAM      50
              221 BOLT    650
              222 BOLT   1250
              231 NUT     700
              232 NUT    1100
      ROLLBACK
 0 0 0 0 0
      CONNECT 'SQLDBA' 'SQLDBAPW'
 0 0 0 0 0
      SQL 'SELECT * FROM INVENTORY'
 0 0 0 0 0    207 GEAR     75
              209 CAM      50
              221 BOLT    650
              222 BOLT   1250
              231 NUT     700
              232 NUT    1100
```

Figure 33. CONNECT command

NOTE: The ROLLBACK is necessary before the CONNECT to disconnect the User ID already active.

FINAL WORDS

This report has demonstrated the basics of using the AP 127 Auxiliary Processor and its associated APL2 workspace, SQL. Most of the AP 127 operations have been covered, along with some functions in the workspace. Using this information, you should be able to start coding SQL statements from APL2. The appendices to this report contain summaries of the commands, workspace functions, error codes, and the SQL statement types.

Of course, there is more to coding an SQL application than just using AP 127 correctly. The SQL database must be designed and put in place. The queries should be written to optimize performance where possible. Locking, authority, and isolation levels are all database parameters that need to be considered in creating a production application. Information on these items is for the most part system-dependent (different for DB2 and SQL/DS), and is available in the reference manuals for the databases. The references section at the end of this report cotains a list of publications that are valuable for those who want to delve further into the SQL issues of the application design.

In addition to finding out more about SQL, you may also want to find out more about some of the APL2 facilities mentioned here. The SQL workspace functions QUERY, EVAL, EVALSIM, CHART, and CHARTDATA were described but not demonstrated. The details on these are contained in "APL2 Programming: Using Structured Query Language". The APL2 Language Reference manual contains information about EACH and the other APL2 operators and functions. See the references section at the back of this report for complete information on these books and other APL2 Manuals.

REFERENCES

APL2 PUBLICATIONS

- Introduction to APL2 (SH20-9229)
- APL2 Programming: Guide (SH20-9216)
- APL2 Programming: Language Reference (GH20-9227)
- APL2 Programming: Using Structured Query Language (SH20-9217)
- APL2 Messages and Codes (SH20-9220)

SQL/DS PUBLICATIONS

- SQL/Data System Application Programming for CMS (GH24-5068)
- SQL/Data System Planning and Administration (SH24-5043)
- SQL/Data System Messages and Codes (GH24-5070)

IBM DATABASE 2 PUBLICATIONS

- IBM Database 2 Introduction to SQL (GC26-4082)
- IBM Database 2 Application Programming Guide for TSO (SC26-4081)
- IBM Database 2 Reference (SC26-4078)
- IBM Database 2 Reference Summary (SX26-3040)
- IBM Database 2 Messages and Codes (SC26-4113)

OTHER PUBLICATIONS

- An Overview of APL2 (GG24-1627)
- Development Guide for Relational Applications (SC26-4130)

APPENDIX A. SQL STATEMENT SUMMARY

STATEMENT TYPE	SQL STATEMENT	AP 127 PROCESSING METHOD
Authorization	GRANT REVOKE	EXEC
Data Definition	CREATE ALTER DROP ACQUIRE	EXEC
Data Manipulation	DELETE INSERT UPDATE	Either: EXEC (no host variables) or PREP,CALL (with host variables)
Query	SELECT	PREP,OPEN,FETCH,CLOSE
Analysis	EXPLAIN	EXEC
Control	LOCK CONNECT COMMIT ROLLBACK	EXEC CONNECT COMMIT ROLLBACK

APPENDIX B. AP 127 OPERATIONS AND SQL WORKSPACE FUNCTIONS

FUNCTION NAME AND SYNTAX	AP 127 OPERATION CODE AND SYNTAX
CALL name [values]	'CALL' name [values]
CHART data	
CLOSE name	'CLOSE' name
COMMIT	'COMMIT'
CONNECT id password	'CONNECT' id password
DESC name	'DESCRIBE' name
EVAL data	
EVALSIM data	
EXEC stmt	'EXEC' stmt
FETCH name [options]	'FETCH' name [options]
GETOPT	'GETOPT'
MESSAGE rcode	'MSG' rcode
NAMES	'NAMES'
OFFER	
OPEN name [values]	'OPEN' name [values]
PREP name stmt	'PREP' name stmt
PURGE name	'PURGE' name
QUE stack	
QUERY name [values]	

FUNCTION NAME AND SYNTAX	AP 127 OPERATION CODE AND SYNTAX
RESUME stack	
ROLLBACK	'ROLLBACK'
SETOPT options	'SETOPT' options
SHOW result	
SQL stmt [values]	
STATE name	'STATE' name
STMT name	'STMT' name
TRACE n1 n2	'TRACE' [n1 n2]
(F UNTIL) stack	

APPENDIX C. RETURN CODE SUMMARY

Return Code Vector	Meaning
0 0 0 0 0	Normal return. All operations completed.
0 0 1 0 0	Normal return, but the result table may not have been completely retrieved.
1 0 0 1 msgn	Error from AP 127. msgn is the number of the AP 127 error message.
1 0 0 2 msgn	Error from DB2 or SQL/DS. msgn is the DB2 or SQL/DS SQLCODE.
1 0 0 3 msgn	Error from the SQL workspace. msgn is the workspace message number.
0 1 0 n msgn	Warning message. n is 1 or 2 as defined here for error returns. msgn is the warning message number.
1 1 0 n msgn	Transaction backout. All changes made to the database since the last COMMIT or ROLLBACK have been discarded. n is 1 or 2 as defined here for error returns. msgn is the error message number.