APL2 EXPLOITING DB2 AND SQL/DS

by James A. Brown
Harlan Crowder

July 1985

TR 03.267

July 1985
TR 03.267

APL2:Exploiting DB2 and SQL/DS

by

James A. Brown
Harlan Crowder

International Business Machines Corporation
General Products Division
Santa Teresa Laboratory
San Jose, California

ABSTRACT

This paper discusses an interactive connection between DB2 or SQL/DS and *APL2*. This connection allows exploitation of the facilities of the database products without the need for any preprocessors or compilers. *APL2* is inherently array oriented and so accesses and processes relational tables all at once -- not row by row. This is particularly significant in light of the fact that *APL2* is not an application that uses relational data but rather a general purpose programming language in which such applications may be written.

APL2: Exploiting DB2 and SQL/DS

by James A. Brown and Harlan Crowder

INTRODUCTION

APL has always had the ability to deal with collections of numbers or collections of characters as single objects. These collections are called arrays. *APL* functions operate on entire collections of data all at once without the need for writing loops. The *APL* operators provide simple control structures for modifying the way functions apply to data.

IBM's recently announced *APL2* Program Product introduces to the *APL* language additional data structures which allow the representation of non-rectangular collections of data. These arrays are called "nested arrays".

The concept of nested arrays and the concept of a relational database evolved independently over the last decade. In light of this, it is surprising that tables from the relational database map so perfectly into nested arrays. This paper will discuss four aspects of the connection between the relational database products and the *APL2* product -- the *APL2* data structures, how they can be used to represent relational tables, accessing DB2 or SQL/DS from *APL2*, and examples of how *APL2* programs can exploit these facilities.

The resulting combination of products is not in itself an end user application but rather represents a powerful tool for the application programmer.

APL2 Data Structures

APL has always provided for storage, display, and computation on simple collections of numbers and simple collections of characters. Here are some examples of this kind of data.

```
      A
3
      B
OPEN THE POD-BAY DOOR, HAL
      C
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
```

The value of *A* is the single number 3. The value of *B* is a string of characters. Such a character string is called a vector. The value of *C* is a table of numbers having three rows and five columns. Such a table is called a matrix.

By default, when you ask for the value of a variable (like *A*, *B*, or *C*), you get the values displayed in a neat arrangement as shown. In general, however, you cannot tell exactly what the structure is. A one row matrix may look very much like a vector. For this reason, *APL2* provides a function (a program, if you will) called *"DISPLAY"*. When this function is applied to data it produces a picture that shows the structure of the data.

Here is the *"DISPLAY"* function as applied to the three variables we saw before:

```
      DISPLAY A
 3
      DISPLAY B
┌→──────────────────────┐
|OPEN THE POD-BAY DOOR, HAL|
└──────────────────────┘
      DISPLAY C
┌→──────────────┐
↓ 1  2  3  4  5|
| 6  7  8  9 10|
|11 12 13 14 15|
└──────────────┘
```

A single value, like *A*, has no structure and so only the value is shown. The vector of characters *B*, however, does have some structure. It is a list of characters arranged in some particular order. This structure is shown by a box around the data with an arrow on the top edge to show that the data is arranged along one direction. The matrix *C* is pictured with an arrow on the top edge and an arrow on the

left edge showing that the data is arranged along two directions.

APL2 has added some new data structures. In *APL2*, we can have an arrangement of data where at any spot in the arrangement is a single number, a single character, or any other data structure.

Here's one example of a new *APL2* data structure:

```
     EMP_DATA
Abe Adams       1234  10 19 42
Bob Blue        2431  11 17 53
Charles Curtis  3142  12 25 39
```

This is a matrix having three rows -- each representing an employee, and three columns -- representing employee names, id numbers, and birth dates. Such data is called a nested array in *APL2*.

The fact that this array has three rows seems pretty obvious. The fact that it has three columns is questionable -- it really doesn't look like three columns. We can use the "*DISPLAY*" function to show the structure of this array.

```
      DISPLAY EMP_DATA
┌→──────────────────────┐
↓ ┌──────────┐  ┌────┐  ┌──────────┐ │
  │Abe Adams │  1234  │10 19 42│ │
  └──────────┘  └────┘  └──────────┘ │
  ┌──────────┐  ┌────┐  ┌──────────┐ │
  │Bob Blue  │  2431  │11 17 53│ │
  └──────────┘  └────┘  └──────────┘ │
  ┌──────────────┐  ┌────┐  ┌──────────┐ │
  │Charles Curtis│  3142  │12 25 39│ │
  └──────────────┘  └────┘  └──────────┘ │
└~──────────────────────┘
```

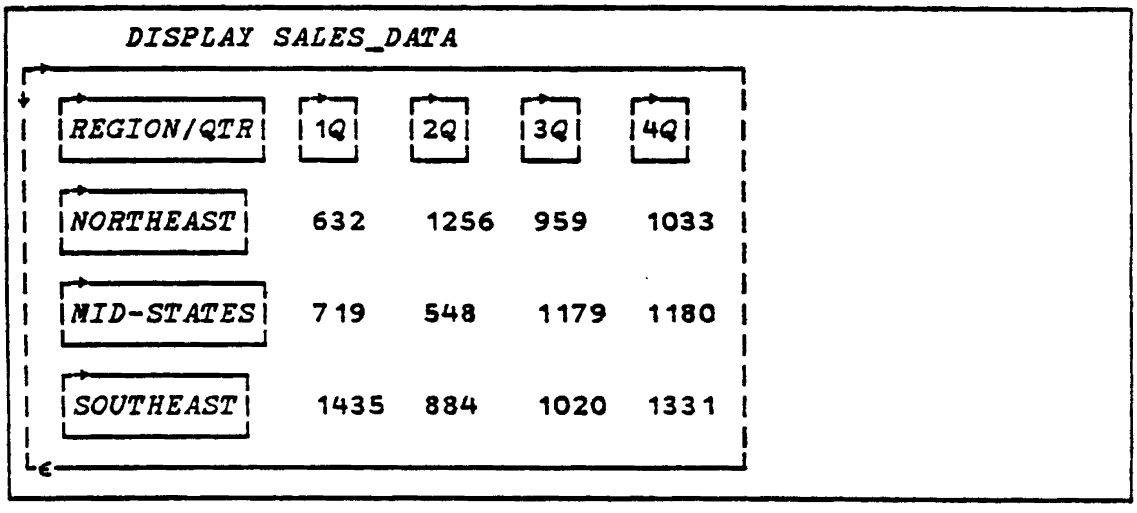
The outer box has an arrow on the top edge and one the left edge and so it is indeed a matrix. Each item in column 1 is a box with one arrow on the top edge and so is a character string. Each item in column 2 has no box and so is a single number. Each item in column three has a box with an arrow and is a numeric vector. (The ~ on the bottom edge means numeric.)

Here's another example of a nested array. This time each item in row one and column one is a character string and every other item is a single number.

```
      SALES_DATA
REGION/QTR    1Q    2Q    3Q    4Q
NORTHEAST    632  1256   959  1033
MID-STATES   719   548  1179  1180
SOUTHEAST   1435   884  1020  1331
```

There are two things to notice about this array. First it is an ordinary 4 by 5 *APL2* matrix. Second, it looks suspiciously like a table from a relational database.

A relation is like a matrix. A column in a relation may be made up of numbers or it may be made up of character strings; an item of data can be missing; and columns have names. Thus "*SALES_DATA*" is an *APL2* matrix that is a representation of a relational table. A table could also be represented other ways and in a moment we'll show another representation. Here's the "*DISPLAY*" of the same array:



This should, by now, be a familiar picture of an *APL2* matrix.

Relational Tables

A relational table is a matrix where rows present data about one entity, columns have one kind of data for each entity, and columns have names. We have already shown that such a table can be represented as a nested array. The representation we used before has the titles as the first row of a matrix. This is one of many ways that a relational

table can be represented and is convenient if only a report is required. If, on the other hand, a computation is required, then that computation is normally done only on the data in a column and not on the title. Therefore we will use another representation of a relation where the titles are stored separately from the data.

For our example, let's assume we have a company called "Hacker's International". We are going to use a set of relational tables to keep track of the personnel and the organization. The first table needed is an employee table which we'll call the Hacker's International staff table and give it the name "*HISTAFF*".

```
     HISTAFF
ID NAME INT DEPT YEARS SALARY   101 INGRAM   ND D01   2 18000
                                102 KAHAN    BA D03   6 32000
                                103 GALVIN   JE D04   5 27000
                                104 BANKS    JA D04  15 35000
                                105 NULVEY   JS D04   3 21000
                                106 DEAN     RA D02  12 38000
                                107 CROW     PJ D02   6 24000
                                108 EATON    FA D03  18 40000
                                109 FARR     JJ D01  25 50000
                                110 HARVEY   HP D04  23 45000
                                111 LAMAR    WJ D02  21 45000
                                112 NELSON   AB D04   7 32000
                                113 ADAMS    SA D01  12 36000
                                114 JACKSON  MA D02   1 16000
```

This table, in *APL2* terms, is a two item vector. The items are the column headings and the data. The *APL2* function, pick (⊃), can be used to select one or the other of these two items. Let's look at this data structure in detail.

First let's look at the first item -- the column titles.

```
      1⊃HISTAFF
ID NAME INT DEPT YEARS SALARY

      ⍴ 1⊃HISTAFF
6

      ⍴¨ 1⊃HISTAFF
 2  4  3  4  5  6
```

The first expression selects the first of the two items from *HISTAFF*. If we ask for its shape (⍴), it tells us that there are six column titles. Each title is a character vector. If we ask for the shape of each item (⍴¨), it tells

us that the first title is a two-element vector, the second is a 4-element vector, etc.

The second item is the data portion of the two item vector.

```
     2⊃HISTAFF
101 INGRAM   MD D01  2 18000
102 KAHAN    BA D03  6 32000
103 GALVIN   JE D04  5 27000
104 BANKS    JA D04 15 35000
105 MULVEY   JS D04  3 21000
106 DEAN     RA D02 12 38000
107 CROW     PJ D02  6 24000
108 EATON    FA D03 18 40000
109 FARR     JJ D01 25 50000
110 HARVEY   HP D04 23 45000
111 LAMAR    WJ D02 21 45000
112 NELSON   AB D04  7 32000
113 ADAMS    SA D01 12 36000
114 JACKSON  MA D02  1 16000
```

This is a nested array with a structure that should look familiar.

The two item vector can be turned into something that looks more like a report with the following expression:

```
     ↑,[1]/HISTAFF
 ID NAME     INT DEPT YEARS SALARY
101 INGRAM   MD  D01      2  18000
102 KAHAN    BA  D03      6  32000
103 GALVIN   JE  D04      5  27000
104 BANKS    JA  D04     15  35000
105 MULVEY   JS  D04      3  21000
106 DEAN     RA  D02     12  38000
107 CROW     PJ  D02      6  24000
108 EATON    FA  D03     18  40000
109 FARR     JJ  D01     25  50000
110 HARVEY   HP  D04     23  45000
111 LAMAR    WJ  D02     21  45000
112 NELSON   AB  D04      7  32000
113 ADAMS    SA  D01     12  36000
114 JACKSON  MA  D02      1  16000
```

All this expression does is catenate (,) the titles as a new row on the data portion.

If a fancier report is wanted, you can write whatever ycu want in *APL*. Here's an example of a slightly fancier report.

```
     PRESFORM HISTAFF
 ID NAME     INT DEPT YEARS SALARY
 == ====     === ==== ===== ======
101 INGRAM   MD  D01      2  18000
102 KAHAN    BA  D03      6  32000
103 GALVIN   JE  D04      5  27000
104 BANKS    JA  D04     15  35000
105 MULVEY   JS  D04      3  21000
106 DEAN     RA  D02     12  38000
107 CROW     PJ  D02      6  24000
108 EATON    FA  D03     18  40000
109 FARR     JJ  D01     25  50000
110 HARVEY   HP  D04     23  45000
111 LAMAR    WJ  D02     21  45000
112 NELSON   AB  D04      7  32000
113 ADAMS    SA  D01     12  36000
114 JACKSON  MA  D02      1  16000
```

The function "*PRESFORM*" (meaning PRESentation FORM) does essentially what the previous expression does except that it puts a little decoration between a title and the column it heads. Here is the definition of the "*PRESFORM*" function:

```
[0]   Z←PRESFORM T
[1]  ⍝ TABLE PRESENTATION FORMAT
[2]   Z←⊃Z⍪((⍴⍕Z←↑T)⍴⍕'=')
[3]   Z←Z,[1]2⊃T
```

Line 2 puts a vector of equal signs under each head, and line 3 catenates this combination as two new rows on the data table.

Communicating with the database system

We'll take a look at how *APL2* communicates with DB2 or SQL/DS by taking the "*HISTAFF*" table and storing it in the database and then doing some selections on it.

Here's how to create the database:

```
      HISTAFF_C
CREATE TABLE HISTAFF
 (ID       SMALLINT,
  NAME     VARCHAR(8),
  INT      CHAR(2),
  DEPT     CHAR(3),
  YEARS    SMALLINT,
  SALARY   INTEGER)
IN APLCLASS

      SQLX HISTAFF_C
```

"*HISTAFF_C*" is just an *APL2* matrix of characters. It looks like a SQL CREATE statement. The function "*SQLX*" is used to pass this matrix to the database system. Since no error report is generated, the database accepted the request and created the database.

The function "*SQLX*" is listed at the end of this paper.

Now we have created the database but it contains no data. Here's how to put data into the table:

```
      HISTAFF_I
INSERT INTO HISTAFF
(ID, NAME, INT, DEPT, YEARS, SALARY)
VALUES(:1, :2, :3, :4, :5, :6)

      SQLX HISTAFF_I (2⊃HISTAFF)
```

Again "*HISTAFF_I*" is just an *APL2* character matrix but, again, it looks something like a SQL command. This time we use the "*SQLX*" function to pass two things to the database: the insert statement and the data portion of our relation.

If we look at the INSERT command, we'll see that this does not look exactly like it would in other languages. This is because *APL2* always tries to deal with whole arrays at once -- not single items or single rows. In other languages the INSERT statement would have names of variables (called host language variables) where we have :1, :2, etc.

This expression causes the entire table to be inserted into the database in what appears to the user as one operation.

Now that we have created the database and stored some data in it, that information is now available to other uses of the database. They can access it any way they want

(assuming they have permission to do so). We can access it using SQL SELECT statements as follows:

```
      S
SELECT ID, NAME, YEARS, SALARY
  FROM HISTAFF
  WHERE YEARS < 6
   AND SALARY > 15000

      SQLX S
 101 INGRAM  2 18000
 103 GALVIN  5 27000
 105 MULVEY  3 21000
 114 JACKSON 1 16000
```

Here "*S*" is a character matrix which represents a SQL SELECT statement. Passing this to the database, with the "*SQLX*" function, gives us back a sub-table as an answer. This sub-table is just a nested matrix to *APL2* and anything you want to do with it you can do. Note that, like the INSERT statement of the previous example, the entire array comes back at once. There is no need to write a loop that causes the table to fetched one row at a time. The concept of a cursor (as used by other host languages) is virtually unneeded in *APL2*. You make a selection request and you get back the answer all at once. For selections that produce very large tables there is an option that allows you to specify some maximum number of rows to produce at one time but, in many practical applications, this facility is not required.

Exploiting the database system

You've now seen much of the facilities of SQL demonstrated. Basically, everything you might want to do to a relational table or a set of relational tables, you can do interactively from *APL2*. Of course much of what you've seen could be done with any of several other products that support the database products.

Next we'll look at some of the ways we can exploit the fact that we have *APL2* which, in addition to being a powerful computing language, also interfaces to many other strategic products.

Producing Charts

One thing you might want to do with relational data is produce business charts. The Interactive Chart Utility (ICU), a part of GDDM, provides a set of menus which aid in the production and tailoring of charts. *APL2* provides a way to call ICU with chart data obtained from a relational table.

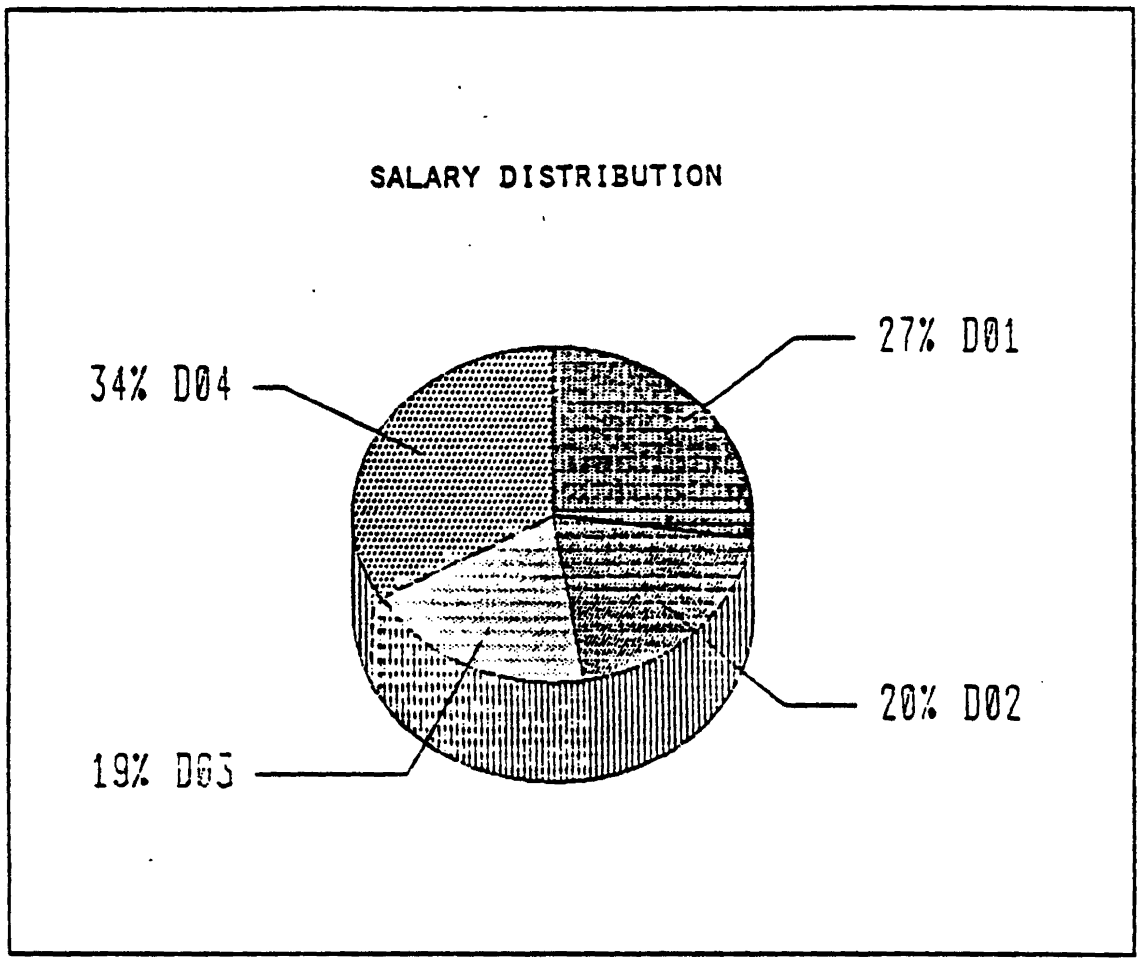
The following SQL query obtains total salary by department, then uses the function *SQLICU*, listed in the appendix, to call the Interactive Chart Utility:

```
      SUMSAL
SELECT SUM(SALARY), DEPT
  FROM HISTAFF
  GROUP BY DEPT

      SDATA←SQLX SUMSAL
      SDATA
D01 104000
D02  78000
D03  72000
D04 128000

      TITLE←'SALARY DISTRIBUTION'
      SQLICU TITLE SDATA
```

The columns of *SDATA* are used as the label and y-axis data in the chart call. After *SQLICU* is invoked, we enter the home panel of ICU, from which we can select various options, such as the display of a pie chart. On GDDM release 4, by specifying the thickness of the pie, we can produce a 3-D display:



Producing Form Letters

For our next example let's look again at Hackers International. We have already seen one table that the company might use -- the employee table *HISTAFF*.

Here's two more tables that this company might keep.

DEPT	*DEPTNAME*	*MGRID*	*LOCATION*
D01	*Administration*	109	*Dallas*
D02	*Production*	106	*Boston*
D03	*Research*	108	*Boulder*
D04	*Marketing*	104	*New York*

The first table relates department numbers to department names and we'll call it *HIDEPT*. The second relates manager identification numbers with location names and we'll call it *HIMGR*. Notice that these three tables have some columns in common. For example, *HISTAFF* and *HIDEPT* have a department name in common. We'll make use of this in a moment.

This defines the database for the company. Let's suppose that the president, Joseph Blow, wants to send a form letter to each employee of the company. How can he make use of the database to do this?

Here's a sample of the letter he wants to send:

```
April 1, 1984
From: Joseph Blow
      CEO, Hackers International
      Silicon Valley, Ca
  To: SA ADAMS
      Administration Department
      Dallas

Just a little note to tell you that
Hackers International grew another 400%
in 1Q84. Our success was really helped
along by your efforts in Dallas.
Keep hacking!
Regards, JB
```

We need three things to get this job done:

- the right information from the database
- a prototype letter
- a file that can be processed to print the individual letters

To begin with let's select the data that we need. Here's a SELECT statement that will do the job:

```
      J
SELECT NAME, INT, DEPTNAME, LOCATION
  FROM     HISTAFF, HIDEPT
  WHERE    HISTAFF.DEPT=HIDEPT.DEPT
  ORDER BY NAME
```

This SELECT statement gets the information we need for each employee: his name and initials, his department name and his

location. This information in not all in one table so the FROM clause mentions two tables: *HISTAFF* and *HIDEPT*. The WHERE clause requests that these two tables be combined on equal department names -- that is, the selection is a join. Finally, it requests that the results be sorted in alphabetical order by name. This sort could have been done in *APL2* after the selection but the data base will do it for us.

Here's what we get out of this selection:

```
      □←MAIL←SQLX J
      MAIL
ADAMS    SA Administration Dallas
BANKS    JA Marketing      New York
CROW     PJ Production     Boston
DEAN     RA Production     Boston
EATON    FA Research       Boulder
FARR     JJ Administration Dallas
GALVIN   JE Marketing      New York
HARVEY   HP Marketing      New York
INGRAM   ND Administration Dallas
JACKSON  MA Production     Boston
KAHAN    BA Research       Boulder
LAMAR    WJ Production     Boston
MULVEY   JS Marketing      New York
NELSON   AB Marketing      New York
```

Notice that, in addition to making the selection, we stored it in the variable "*MAIL*" (and then requested that the value be printed). Now we have an ordinary *APL2* array that we can process as we wish.

The next piece of information that we need is a prototype letter. Here it is:

```
.nf
.ll 40
April 1, 1984
From: Joseph Blow
      CEO, Hackers International
      Silicon Valley, Ca
  To: &INT. &NAME
      &DEPTNAME. Department
      &LOCATION
.sk
Just  a little  note to  tell you  that
Hackers International grew another 400%
in 1Q84.  Our success was really helped
along by your efforts in &LOCATION..
.sk;Keep hacking!
.sk;Regards, JB
.pa
```

This prototype letter is just an ordinary sequential file stored on the host system. The details of the *APL2* functions used are in the Appendix but if you want to produce the letters, here's what you enter and a few lines of what you get:

```
      '.im letter' SETFILE MAILHEAD MAIL
.se NAME = 'ADAMS
.se INT = 'SA
.se DEPTNAME = 'Administration
.se LOCATION='Dallas
.im letter
.se NAME = 'BANKS
.se INT = 'JA
.se DEPTNAME = 'Marketing
.se LOCATION='New York
.im letter
.se NAME = 'CROWS
.se INT = 'PJ
.se DEPTNAME = 'Production
.se LOCATION='Boston
.im letter
```

You may recognize this as a SCRIPT file suitable for processing by the Document Composition Facility. Thus *APL2* has tied SQL to SCRIPT!

This SCRIPT file could now be processed to produce the set of letters as required.

Conclusion

APL2 has access to the relational database products and many other products. We have shown examples of how *APL2* can be used to produce business charts using the Interactive Chart Utility, and form letters by using the Document Composition Facility.

In addition, because *APL2* interfaces to ISPF, you could build a full screen panel application to be used in the building and updating of a database.

Thus *APL2* has the ability to tie together all the products, to which it interfaces, in a unified package.

References

- SH20-9216 APL2 Programming: Guide
- SH20-9217 APL2 Programming: Using SQL
- SH20-9218 APL2 Programming: System Services Reference
- SH20-9227 APL2 Programming: Language Reference
- SH20-9229 An Introduction to APL2
- SC26-4081 DB2 Application Programming Guide
- GH24-5065 SQL/DS Concepts and Facilities for VM/SP
- SH24-5068 SQL/DS Application Programming for VM/SP
- SC33-0102 GDDM PGF Programming Reference
- SC33-0111 GDDM PGF Interactive Chart Utility User's Guide
- SH35-0070 DCF SCRIPT/VS Language Reference

Acknowledgement

The authors wish to thank Ed Eusebi who produced the chart example.

Appendix:

APL2 Functions

SQLX is the cover function used in the paper for communication with the database system. It merely passes the request to the IBM supplied cover function (in distributed workspace SQL) and strips off the return code before returning the result. This allows us to ignore error conditions during the discussion of the SQL interface.

```
     ∇Z←SQLX SQL_STMT
[1] ⍝ SQL cover function
[2]  Z←SQL SQL_STMT ⍝ Pass request to IBM cover function
[3]  →(0∨.≠↑Z)/0    ⍝ Return everything if error
[4]  Z←2⊃Z          ⍝ ELSE  only return data
```

The *SETFILE* function takes a relational table and builds a SCRIPT file. The right argument to the function is a relation in the form used in the paper -- a two item vector with column titles first and the data matrix second. The left argument is the header line to be appended above the information produced for each row of the table.

Line 2 turns each item of the relation into a character string even if it was a number originally. (The example we gave in the paper did not need this because we only selected character columns.) Line 3 builds each ".se" line up to and including the open quote given before the value. Note that the column names are used as the set symbol names. Line 4 attaches to these lines the variable data from each row of the relation and then builds the simple matrix from it.

```
     ∇Z←P SETFILE HD;D;H
[1] ⍝ Generate DCF set symbols
[2]  D←⍕¨2⊃HD                ⍝Turn each item into characters
[3]  H←(⊂'.se '),¨(↑HD),¨⊂' = ''' ⍝Build line
[4]  Z←⊃[2],(((⍴D)⍴H),¨D),⊂P ⍝Return simple character matrix
```

The following function is the one used to call the Interactive Chart Utility. The right argument is a two-item vector; a character string which represents the title, and a 2-column SQL table. This function can be changed to use X-axis data, multiple Y-axes, and keys. For an explanation of the values used in the call, see the description of AP126 in *APL2* Systems Services, and of ICU in GDDM PGF Programming Reference.

SQLICU uses the IO function from the *APL2* distributed workspace 1 *UTILITY*, to convert integers to System 370 characters.

```
     ∇Z←SQLICU TLY;LABELS;TITLE;X;Y;CCTL;DAT;CTL;⎕IO;R
[1]   ⎕IO←1
[2]  ⍝ INVOKE INTERACTIVE CHART UTILITY
[3]   (TITLE LABELS Y)←(1↑TLY),⊂[1]2↓TLY
[4]   LABELS←⊂[2]↓LABELS          ⍝ LABELS SAME LENGTH
[5]   X←⍳⍴Y                       ⍝ DEFAULT X-AXIS
[6]  ⍝ OFFER TO SHARE WITH AP126
[7]   ⎕ES(∨/2≠126 ⎕SVO¨'CTL' 'DAT')/'AP126 SHARE ERROR'
[8]  ⍝ BUILD CHART CONTROL...
[9]   CCTL←''
[10]  CCTL←CCTL,4 IO 0            ⍝ LEVEL     0=OLD LEVEL
[11]  CCTL←CCTL,4 IO 1            ⍝ DISPLAY   1=HOME PANEL,2=CHART
[12]  CCTL←CCTL,4 IO 1            ⍝ HELP      1=DISPLAY PFKEY INFO
[13]  CCTL←CCTL,4 IO 0            ⍝ ISOLATE   0=ALL FACILITIES
[14]  CCTL←CCTL,'*        '       ⍝ FORMNAME  *=DEFAULT
[15]  CCTL←CCTL,'*        '       ⍝ DATANAME  *=OTHER PARMS
[16]  CCTL←CCTL,4 IO 0            ⍝ BINDING   0=TIED,1=FREE
[17]  CCTL←CCTL,,4 IO,1           ⍝ DATA GROUPS
[18]  CCTL←CCTL,,4 IO,-⍴X         ⍝ ELEMENTS - WITH LABELS
[19]  CCTL←CCTL,4 IO 0            ⍝ KEYL      0=NO KEYS
[20]  CCTL←CCTL,,4 IO⍴↑LABELS     ⍝ LABELL    0=NO LABELS
[21]  CCTL←CCTL,,4 IO⍴TITLE       ⍝ HEADINGL  0=NO HEADING
[22]  CCTL←CCTL,'*        '       ⍝ PRTNAME   *=UNKNOWN
[23]  CCTL←CCTL,4 IO 0            ⍝ PRTDEP    0=DEFAULT
[24]  CCTL←CCTL,4 IO 80           ⍝ PRTWID    0=DEFAULT
[25]  CCTL←CCTL,4 IO 1            ⍝ PRTCOPY   0=DEFAULT
[26]  ⍝ SPEC CHART-CONTROL,KEYS,LABELS,TITLE
[27]  DAT←CCTL,'',(∊LABELS),TITLE
[28]  ⍝ SPEC NUMERIC INFORMATION
[29]  R←(⍴CCTL),1 0,(⍴,X),X,(⍴,Y),(,Y),0,(⍴∊LABELS),⍴TITLE
[30]  CTL←¯10,R                   ⍝ CALL ICU
```