APL2 and SQL: A Tutorial

August, 1989

Nancy Wheeler

IBM
APL Development
Santa Teresa Laboratory
San Jose, California

Reprinted with permission from the proceedings of APL89, August 1989, New York, New York. Copyright 1989, Association for Computing Machinery, Inc.

Contents

Part 1: SQL 1
Relational Data 1
SQL Statements 2
  Data Definition 2
  Data Manipulation 3
  Data Retrieval 4
  Authorization Statements 5
  Control Statements 5
  Analysis 6
Using SQL 7
  Using SQL from Programs 7
  Static SQL vs. Dynamic SQL 7

Part 2: APL2 8
Relational Data in APL2 8
SQL Statements in APL2 9
  Data Definition, Authorization 10
  Data Manipulation 10
  Data Retrieval 11
  Control Statements 17
  EXPLAIN 19
  Error Handling 19

Part 3: APL2 and SQL 23
Locking 23
  Lock Size 23
  Isolation Level 23
  Explicit Lock Control 24
Authority 25
The Index 25
Storage 26
Performance 27

Appendix A. SQL Statement Summary 28

Appendix B. AP 127 Operations and SQL Workspace Functions 29

Appendix C. Return Code Summary 31

Appendix D. Installation Instructions 32
SQL/DS (CMS) Environment 32
DB2 (TSO) Environment 33

Appendix E. References 35
DB2 Publications 35
SQL/DS Publications 35
APL2 Publications 35

Other 35

Figures

1. Relational Table 1
2. CREATE TABLE statement 2
3. Data Manipulation Statements 4
4. Data Retrieval Statements 4
5. Authorization Statements 5
6. Control Statements 6
7. Relational data in an APL2 matrix 8
8. Relational data in an APL2 vector 9
9. Data Definition Statements from APL2 10
10. Data Manipulation Statements in APL2 11
11. Simple Data Retrieval 12
12. DESCRIBE 13
13. Data Retrieval with variable substitution 14
14. SETOPT command 15
15. Vector Format with Length Matrix 15
16. Fetching a Table in Pieces 16
17. Null Result Tables 17
18. ROLLBACK, COMMIT 18
19. CONNECT command 18
20. EXPLAIN command 19
21. An AP 127 error 19
22. An SQL error 20
23. Query Cost Estimate 22
24. Forms of the ISOL command 24
25. PURGE command 27

Part 1: SQL

Before beginning the discussion of SQL, it is necessary to define some terms.

- SQL (Structured Query Language) is a language defined to access relational databases. The SQL language has been implemented by various vendors in support of their relational databases, and there is an ANSI standard for the SQL language.
- SQL/DS (SQL/Data System) is IBM's relational database system in the VM environment.
- DB2 (IBM Database 2) is IBM's relational database system in the MVS environment.

Throughout this tutorial, the term SQL will be used to refer to the SQL language. Except where explicitly stated, the uses and examples of SQL seen here are supported both by SQL/DS and DB2.

Relational Data

Simply stated, relational data is data that is arranged in a two-dimensional tabular format. The columns of the matrix have attributes that define the type of the data, and the rows contain the data. The intersection of a row and column is called a field or value. Figure 1 shows an example of a relational table.

```
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 1. Relational Table

One of the main advantages of using relational databases is that SQL users need not know how relational data is stored in the computer in order to use it. Using SQL statements, the user tells the database system what is to be done in a conceptual manner, and the database system then accesses the data necessary for the specified task. Thus, even though the underlying operating systems may be completely different in different relational databases (as with SQL/DS and DB2), the user can state the problem in a well-defined, familiar manner.

SQL Statements

For the purposes of this tutorial, we will group the major SQL statements into categories. Complete syntax for all of these, and for additional SQL statements, can be found in the database reference manuals.

Data Definition

Data definition statements are those that allow you to define your database configuration. The **CREATE** statement has several forms for the creation of tables, indices, views (logical tables), and synonyms (alias names). The **ALTER** statement allows, with some restrictions, the changing of table definitions, and the **DROP** statement is for deletion of entire tables, indices, views and synonyms. Figure 2 shows the **CREATE TABLE** statement used to create the table shown in Figure 1 on page 1.

```
CREATE TABLE ABC
(NAME VARCHAR(20) NOT NULL,
 YOS  SMALLINT,
 LEVEL SMALLINT,
 SALARY INTEGER,
 DEPT VARCHAR(8),
 PROJ CHAR(3))
 IN MYSPACE
```

Figure 2. CREATE TABLE statement

Each column must be given a name and datatype attribute. Table 1 is a summary of the datatypes currently supported by SQL/DS and DB2. The **NOT NULL** attribute is optional, and indicates to SQL that an error should be given if an attempt is made to add a row to the table with data missing for the indicated column.

SQL Datatype	Description
INTEGER	Fullword (31-bit) signed integer
SMALLINT	Halfword (15-bit) signed integer
FLOAT	Double Precision (8-byte) floating point. May also be indicated as DOUBLE PRECISION or FLOAT(n) where $22 \le n \le 53$.
REAL	Single Precision (4-byte) floating point. May also be indicated as FLOAT(n) where $1 \le n \le 21$.
DECIMAL(p,s)	Decimal number, where p indicates precision (1 to 15) and s indicates scale (0 to p)
CHAR(n)	Fixed-length character string of length n, where $1 \le n \le 254$. Data is padded with blanks if smaller than the defined size.
VARCHAR(n)	Varying-length character string, with maximum length n, where $1 \le n \le 32767$.

Table 1 (Part 1 of 2). SQL Datatypes

SQL Datatype	Description
LONG VARCHAR	Varying-length character string, with maximum length 32767.
GRAPHIC(n)	Fixed-length graphic (double-byte) string of length n, where $1 \le n \le 127$. Data is padded with blanks if smaller than the defined size.
VARGRAPHIC(n)	Varying-length graphic string, with maximum length n, where $1 \le n \le 16383$.
LONG VARGRAPHIC	Varying-length graphic string, with maximum length 16383.
DATE	A date in one of four standard forms (ISO, USA, JIS, EUR).
TIME	A time in one of four standard forms (ISO, USA, JIS, EUR).
TIMESTAMP	A date and time combination in ISO form.

Table 1 (Part 2 of 2). SQL Datatypes

If you want to attach additional information to tables, views or columns, the **LABEL ON** and **COMMENT ON** statements may be used after the table or view has been created. Column labels are useful if you find the 18-character length limit on column names too restrictive, and they can be retrieved along with or instead of the names when you retrieve the data. Comments are simply placed in the system catalog tables for informational purposes.

The final clause on the **CREATE TABLE** statement indicates where in the database to place the table. An assumption is made that the entity ***MYSPACE*** exists and that the creator of the table is authorized to use it. This is one area where the individual database implementations are not the same. The allocation of space in the database for the table is handled differently in DB2 and SQL/DS and requires some system-dependent information. The process of acquiring the space is usually done once, when a new application is designed, with the help of a database administrator, and will not be discussed further here.

Note that this example shows only the simplest form of **CREATE TABLE**. Later releases of SQL/DS and DB2 have added support for referential integrity, which allows association between tables and requires a more detailed form.

Data Manipulation

The three data manipulation SQL statements are **INSERT**, **UPDATE** and **DELETE**. They operate on individual rows of a table, and their functions are indicated by their names.

The **UPDATE** and **DELETE** statements choose the row operated on in one of two ways. The first is to use a search condition (**WHERE** clause). The number of rows changed or deleted depends on the number of rows that meet the search criteria. The second method is to position a data retrieval cursor at the row of the table to be operated on and use the **CURRENT OF** ***cursorname*** clause. In this case, the number of rows changed or deleted is always one. Cursors will be discussed in more detail later in this tutorial.

Figure 3 on page 4 shows some examples of data manipulation statements. Note the use of the keyword NULL to represent the absence of data.

```
INSERT INTO ABC VALUES('KAHAN',6,7,32000,'RES','PR2')

INSERT INTO ABC VALUES('JACKSON',1,1,16000,'PROD',NULL)

UPDATE ABC SET SALARY=35000 WHERE NAME='KAHAN'

UPDATE ABC SET SALARY=35000 WHERE CURRENT OF MYCURSOR

DELETE FROM ABC WHERE DEPT='RES'
```

Figure 3. Data Manipulation Statements

Data Retrieval

The data retrieval SQL statement is **SELECT**. The search conditions are specified in the **WHERE** clause of the statement. The **WHERE** clause can range in complexity from being non-existent to containing an entire other **SELECT** statement. You can select all or any subset of columns from one or multiple tables.

Another feature of the **WHERE** clause is that some values may be left as variables, to be filled in at execution time. This is handled differently in different languages, but most use the colon (:) to identify the variable.

The search capabilities built into the SQL language are very powerful. It would be impossible to cover them in depth in this short tutorial. Some basic examples are given in Figure 4, and the reader is referred to the database documentation for the more advanced forms.

```
SELECT * FROM ABC

SELECT * FROM ABC ORDER BY DEPT

SELECT NAME,DEPT,PROJ FROM ABC

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

SELECT NAME,SALARY FROM ABC WHERE DEPT = :myvariable
```

Figure 4. Data Retrieval Statements

When a **SELECT** statement is issued, the result is a new relational table. (This explains the ability to select within a select).

Associated with the **SELECT** statement is the concept of a ***cursor***. A cursor is an imaginary pointer that indicates which row of the result table is the ***current*** row. The cursor is set to point to the top of the table, and as each row is retrieved it is moved to point to that row. Once the cursor is pointing to a row, the **CURRENT OF** clause may be used on an **UPDATE** or **DELETE** to modify or delete that row. When the cursor moves beyond the end of the table, the program will be signalled that there is no more data in the table. More details on how the cursor is created and used will be given in Part 2.

Authorization Statements

Once a user or application has created some relational tables, it is usual that some other users or applications will want to use that data. The authorization statements **GRANT** and **REVOKE** allow specification of which users can perform which operations on data created by another user. See Figure 5 on page 5 for some examples of authorization statements.

```
GRANT ALL ON ABC TO USER1

GRANT INSERT, UPDATE (PROJ,DEPT) ON ABC TO USER2

GRANT SELECT ON ABC TO PUBLIC

REVOKE INSERT ON ABC FROM USER2
```

Figure 5. Authorization Statements

Control Statements

Control statements do not operate on specific data, but rather specify the boundaries of a ***unit of work*** or a lock held on data.

Units of work in SQL are sequences of operations which are performed together. Only when all the operations in the sequence are successfully completed is the database to be permanently changed. A unit of work is begun when the first SQL statement is issued to the database, and is terminated by either a **COMMIT** statement, which makes the changes permanent, or a **ROLLBACK** statement, which cancels them. After the **COMMIT** or **ROLLBACK**, a new unit of work is begun. In SQL/DS, the additional **RELEASE** option causes the database connection to be completely severed.

When a user begins a new unit of work, a user id is assigned by the database. The default id is normally the operating system user id, but each database system provides a means of overriding that default. In SQL/DS, the **CONNECT** statement allows specification of an alternative id. If the correct password is given, the user may become another user or connect to a special id created just for a particular application. In DB2, the **SET CURRENT SQLID** statement allows the user to change the value of the authorization ID established by the DB2 authorization exits. More information on these exits is available in the DB2 documentation.

Locks on data can be controlled in various ways, one of which is to use the **LOCK** statement. This statement causes the database to override the implicit locking parameters, which may be useful when updates are to be made which depend on the status of the entire table remaining consistent. The lock override stays in effect until the end of the unit of work.

Figure 6 on page 6 shows examples of control statements.

```
SQL/DS and DB2:

ROLLBACK WORK

COMMIT WORK

LOCK TABLE ABC IN SHARE MODE

LOCK TABLE ABC IN EXCLUSIVE MODE


SQL/DS:

COMMIT WORK RELEASE

ROLLBACK WORK RELEASE

CONNECT MYUSER IDENTIFIED BY MYPASSWORD


DB2:

SET CURRENT SQLID MYUSER
```

Figure 6. Control Statements

Analysis

The **EXPLAIN** statement is a special SQL statement that does not do any operations on data, but rather analyzes those operations. It takes an SQL statement as an argument, and places information about path selection into a special SQL table. This can be very useful in tuning an application to take the best possible advantage of indices. For more information on **EXPLAIN**, see the database documentation.

Using SQL

Using SQL from Programs

SQL is not a stand-alone language, in that it does not compile directly into executable machine language modules. It is either embedded in programs written in other compiled languages (such as assembler, FORTRAN, PL/I, COBOL and C), called interactively by interpretive languages (such as APL2, REXX, and BASIC) or supported internally by end-user tools such as QMF. When using it in conjunction with compiled languages, the source code is passed through a pre-processor before the compile which translates the SQL statements into statements in the compiled language. With interpreted languages and end-user tools, there is an special step when the language or tool is installed to initialize the database support, but no pre-process is required when issuing SQL statements.

Static SQL vs. Dynamic SQL

Static SQL is a form of SQL where the SQL statements are known in advance and coded directly into a compiled program. Any variable data is passed in program variables whose names are also coded into the program. This type of SQL is supported only from compiled languages.

With Dynamic SQL, the SQL statements are not known until execution time. They are built into a buffer, and the buffer and any variable data is passed at execution time. This type of SQL is supported both from compiled and interpreted languages, and is the type of SQL we will be discussing in Parts 2 and 3 of this tutorial.

Each of the two types of SQL has advantages and disadvantages. Static SQL is usually faster at execution time, as the path to the data can be determined at compile time. However, if the database status changes (new indices are added, etc.) and the program is not recompiled, this advantage can turn into a disadvantage. Dynamic SQL, when issued from an interpreted language, requires less coding time and is easily modified if the problem changes, with no pre-process or compile required.

Whether to use Static or Dynamic SQL is a choice akin to that of whether to use a compiled or interpreted language. Each has its strengths, and many times a combination of the two provides the optimum solution.

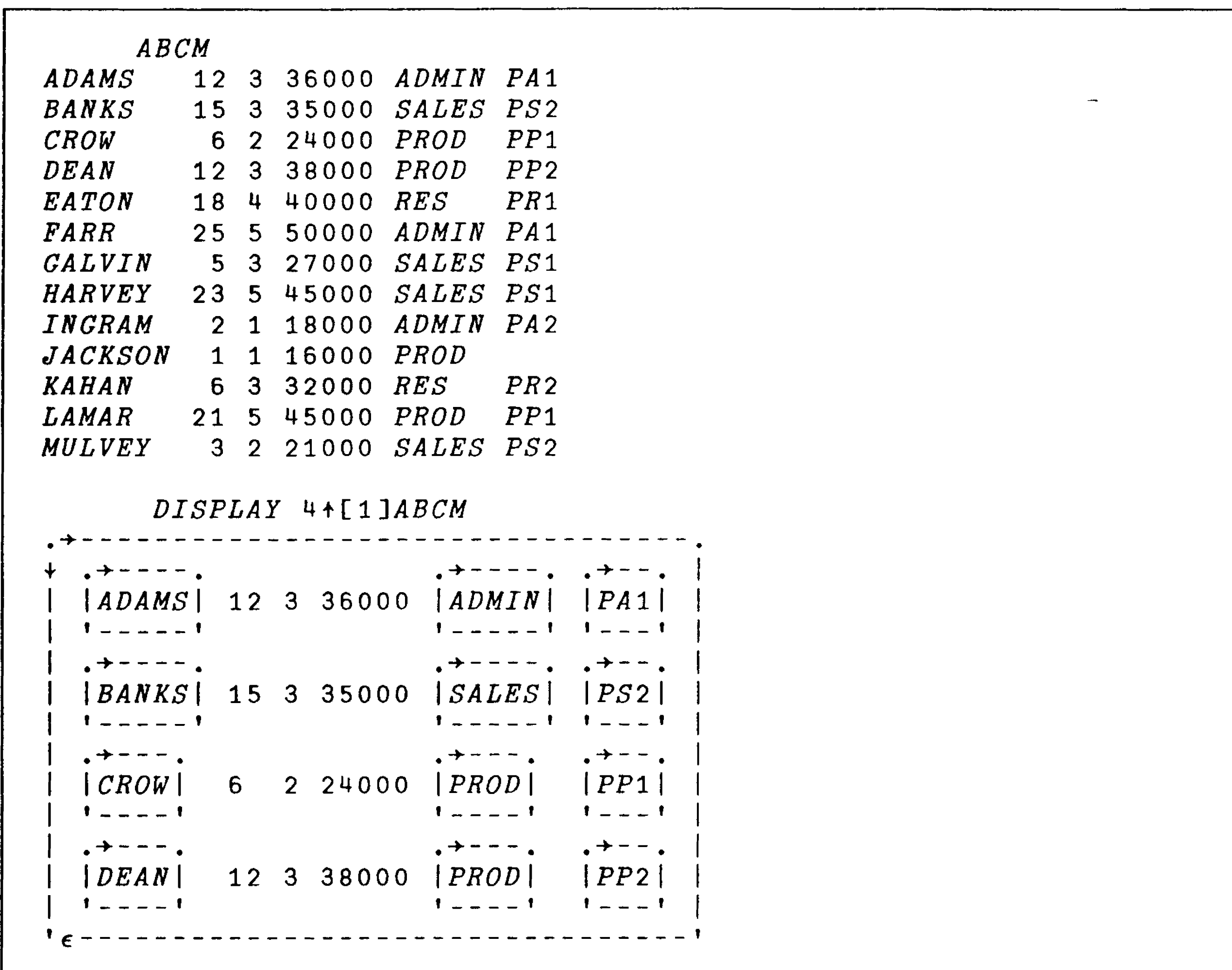
Part 2: APL2

An auxiliary processor, AP 127, allows APL2 programmers to imbed Structured Query Language (SQL) statements in their APL2 functions.

Along with AP 127, a workspace called SQL is distributed with the APL2 product. The SQL workspace contains APL2 functions for using AP 127. It contains a simple function for each AP 127 command, and some additional higher-level functions. In these examples we will use the simple command functions.

Relational Data in APL2

Figure 1 on page 1 showed an example relational table. We could represent that table in an APL2 matrix with a row for each row of the table and a column for each column. If we did, it would look like Figure 7.

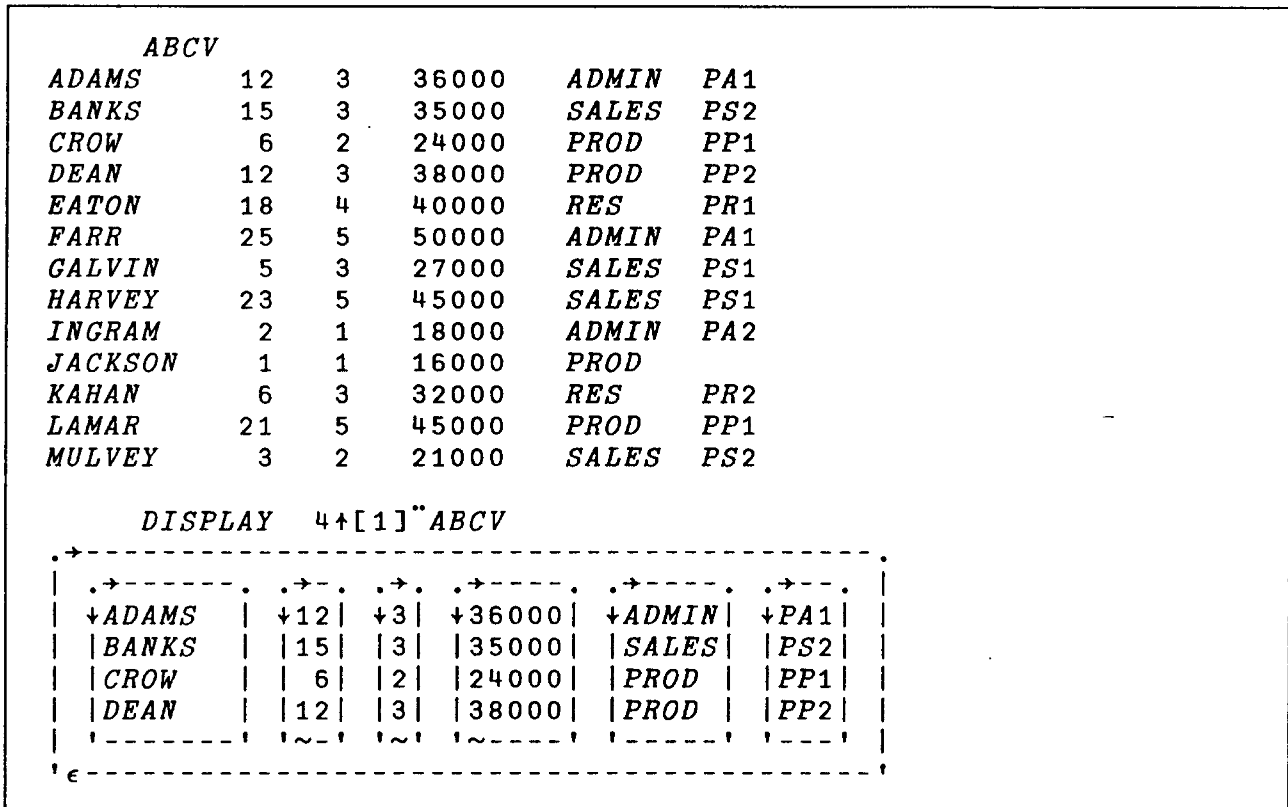


```
      ABCM
ADAMS    12 3 36000 ADMIN PA1
BANKS    15 3 35000 SALES PS2
CROW      6 2 24000 PROD  PP1
DEAN     12 3 38000 PROD  PP2
EATON    18 4 40000 RES   PR1
FARR     25 5 50000 ADMIN PA1
GALVIN    5 3 27000 SALES PS1
HARVEY   23 5 45000 SALES PS1
INGRAM    2 1 18000 ADMIN PA2
JACKSON   1 1 16000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

      DISPLAY 4↑[1]ABCM
.→--------------------------------.
↓ .→----.                .→----. .→--. |
| |ADAMS| 12 3 36000 |ADMIN| |PA1| |
| '-----'                '-----' '---' |
| .→----.                .→----. .→--. |
| |BANKS| 15 3 35000 |SALES| |PS2| |
| '-----'                '-----' '---' |
| .→---.                 .→---.  .→--. |
| |CROW| 6  2 24000 |PROD|  |PP1| |
| '----'                 '----'  '---' |
| .→---.                 .→---.  .→--. |
| |DEAN| 12 3 38000 |PROD|  |PP2| |
| '----'                 '----'  '---' |
'∊--------------------------------'
```

Figure 7. Relational data in an APL2 matrix

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 ABCM 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.

Another possible way to represent a relational table is as a vector of matrices, one for each column. Figure 8 shows that form.



```
      ABCV
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]¨ABCV
.→--------------------------------------------.
| .→------.  .→-.  .→.  .→----.  .→----.  .→--. |
| ↓ADAMS  |  ↓12|  ↓3|  ↓36000|  ↓ADMIN|  ↓PA1| |
| |BANKS  |  |15|  |3|  |35000|  |SALES|  |PS2| |
| |CROW   |  | 6|  |2|  |24000|  |PROD |  |PP1| |
| |DEAN   |  |12|  |3|  |38000|  |PROD |  |PP2| |
| '-------'  '~-'  '~'  '~----'  '-----'  '---' |
'∊--------------------------------------------'
```

Figure 8. Relational data in an APL2 vector

In this alternative form, note that the variable-length character fields must be padded with blanks to the width of the widest value to create a uniform matrix. Null items must also be replaced with their prototype values, which are 0 for numeric items and blank for character items.

APL2 allows retrieval of SQL data in both of these forms.

SQL Statements in APL2

In Part 1, we showed examples of SQL statements. We will now see how to execute those statements from within APL2.

Data Definition, Authorization

Data definition and authorization statements are executed directly, as is, by the database system. No variable substitutions are allowed. For this type of statement, we use the AP 127 command *EXEC*, which expects only one argument, the SQL statement. In Figure 9 on page 10 we will use the *EXEC* function to execute the **CREATE** statement from our example, and to grant authority to all users to retrieve data from it. We have placed the **CREATE** in an APL2 character matrix called *ABCC*, since it is a longer statement.

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

      EXEC ABCC
 0 0 0 0 0

      EXEC 'LABEL ON COLUMN ABC.NAME IS ''THIS IS A LABEL'''
 0 0 0 0 0

      EXEC 'GRANT SELECT ON ABC TO PUBLIC'
 0 0 0 0 0
```

Figure 9. Data Definition Statements from APL2

For those familiar with Dynamic SQL, the *EXEC* command issues an **EXECUTE IMMEDIATE** to SQL.

AP 127 always returns a 5-item numeric return code, followed by any result data. For these statements, there is no result data, so the second item of the result is null. We will explain the meaning of non-zero return codes in a later section. Five zero codes, of course, means that operation completed successfully.

Data Manipulation

Data manipulation statements can be processed in two different ways. If there are no variable substitutions, they can be processed like data definition statements, with the *EXEC* command. If you wish to execute an SQL statement more than once, however, it can be much more efficient to use variable substitution.

With variable substitution in APL2, the places for the values are indicated in the SQL statement by a colon (:) followed by a number which represents an index into an APL2 variable. The statement in that form is passed to AP 127 with a *PREP* command, and a name is attached to the statement to identify it for later execution. AP 127 will remember the indices and replace them with the Dynamic SQL placeholder "?" before issuing an SQL **PREPARE**.

To execute the statement, an AP 127 *CALL* command is issued against the statement name, and the data, or value-list, is passed as an argument to that command. The value-list can contain more items than are actu-

ally indexed, and the indices can select items in any order. The *CALL* command issues a Dynamic SQL **EXECUTE**. It may be repeated as many times as necessary, with different data passed each time.

Figure 10 shows examples of data manipulation statements in APL2.

```
Without variable substitution:

      EXEC 'UPDATE ABC SET SALARY=35000 WHERE NAME=''KAHAN'''
 0 0 0 0 0

      EXEC 'DELETE FROM ABC WHERE DEPT=''RES'''
 0 0 0 0 0

With variable substitution:

      PREP 'NAME' 'INSERT INTO ABC VALUES(:1,:2,:3,:4,:5,:6)'
 0 0 0 0 0

      CALL 'NAME' ('KAHAN' 6 7 32000 'RES' 'PR2')
 0 0 0 0 0

      CALL 'NAME' ('JACKSON' 1 1 16000 'PROD' '')
 0 0 0 0 0
```

Figure 10. Data Manipulation Statements in APL2

Data Retrieval

There are five required steps to perform a SELECT in Dynamic SQL. They are **PREPARE, DESCRIBE, OPEN, FETCH** and **CLOSE**. The AP 127 commands corresponding to these steps are ***PREP*, *OPEN*, *DESCRIBE*, *FETCH*** and ***CLOSE***.

The ***PREP*** command is the same for data retrieval as for data manipulation, except that it is required, whether or not there are variables in the statement. It sends the statement to SQL and assigns a name to it.

The ***DESCRIBE*** command returns information about the result table names, datatypes, null status and labels. This AP 127 command may be issued at any time after the ***PREP***, or may be omitted if not needed by the application. If it is omitted, AP 127 will issue the SQL **DESCRIBE** automatically before the **OPEN** and save the information for use later on. There are four different forms of describe which return the column names and labels in different combinations.

The ***OPEN*** command receives the value-list, if any, and sets the cursor to the beginning of the table (it does not yet point to any rows). The ***FETCH*** command causes the data to be passed from SQL. The ***CLOSE*** command tells SQL that we are done with the cursor.

As with data manipulation statements, the ***PREP*** command may be issued once, with variable indices, and the ***OPEN*, *FETCH*, *CLOSE*** sequence may be repeated, with different data passed on each ***OPEN***.

Figure 11 on page 12, Figure 12 on page 13 and Figure 13 on page 14 show some simple examples of this sequence.

```
      PREP 'NAME' 'SELECT * FROM ABC'
0 0 0 0 0

      OPEN 'NAME'
0 0 0 0 0

      FETCH 'NAME'
0 0 0 0 0   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

      CLOSE 'NAME'
0 0 0 0 0
```

Figure 11. Simple Data Retrieval

```
      PREP 'NAME' 'SELECT * FROM ABC'
0 0 0 0 0

      DESC 'NAME'
0 0 0 0 0    NAME       YOS  LEVEL SALARY   DEPT   PROJ
             V 20       S    S     I        V 8    C 3
             NOT NULL   NULL NULL  NULL     NULL   NULL

      DESC 'NAME' 'LABELS'
0 0 0 0 0    THIS IS A LABEL
             V 20              S    S    I      V 8    C 3
             NOT NULL          NULL NULL NULL   NULL   NULL

      DESC 'NAME' 'ANY'
0 0 0 0 0    THIS IS A LABEL   YOS  LEVEL SALARY   DEPT   PROJ
             V 20              S    S     I        V 8    C 3
             NOT NULL          NULL NULL  NULL     NULL   NULL

      DESC 'NAME' 'BOTH'
0 0 0 0 0    NAME              YOS  LEVEL SALARY   DEPT   PROJ
             V 20              S    S     I        V 8    C 3
             NOT NULL          NULL NULL  NULL     NULL   NULL
             THIS IS A LABEL
```

Figure 12. DESCRIBE

```
      PREP 'NAME' 'SELECT NAME FROM ABC WHERE DEPT = :1'
0 0 0 0 0

      OPEN 'NAME' (⊂'ADMIN')
0 0 0 0 0

      FETCH 'NAME'
0 0 0 0 0    ADAMS
             FARR
             INGRAM

      CLOSE 'NAME'
0 0 0 0 0

      OPEN 'NAME' (⊂'SALES')
0 0 0 0 0

      FETCH 'NAME'
0 0 0 0 0    BANKS
             GALVIN
             HARVEY
             MULVEY

      CLOSE 'NAME'
0 0 0 0 0
```

Figure 13. Data Retrieval with variable substitution

You will notice that each *FETCH* request made to AP 127 returns multiple rows of the SQL table, even though the SQL FETCH only can receive one row at a time. The number of rows returned is controlled by an AP 127 option. AP 127 will return the number of rows you request, or the whole table, whichever is smaller. If the entire table is not retrieved on the first call, the third item of the return code is set to one. You can repeat the *FETCH* step as many times as necessary to retrieve the entire table.

A second AP 127 option allows you to choose the data format. As shown previously, there are two different formats. The default is the first, or ***matrix*** format. The alternate format, which actually consumes less space and so can be more efficient, is called ***vector*** format.

If you choose to use the vector format, you may also request from AP 127 an additional data item called the ***length matrix*** which will tell you the lengths of the data before padding and null replacement. You can use this information to recreate the matrix form of the data, or perhaps to substitute alternate values for the SQL null.

All of these options can be set permanently with the AP 127 *SETOPT* command, or temporarily during execution of the query by passing them as arguments to the *FETCH* command. Figure 14 on page 15 shows the use of the *SETOPT* command, which sets options permanently. Any or all of the options may be set on each call. Note that the *SETOPT* command does not cause any communication with SQL, it simply tells AP 127 how you would like to receive your data.

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

      SETOPT 500
0 0 0 0 0

      SETOPT 'LENGTH'
0 0 0 0 0

      SETOPT 'MATRIX' 'NOLENGTH'  20
0 0 0 0 0
```

Figure 14. SETOPT command

Figure 15 shows the result when the vector format and length matrix options are in effect.

```
      PREP 'NAME' 'SELECT NAME,PROJ FROM ABC'
0 0 0 0 0

      OPEN 'NAME'
0 0 0 0 0

      DISPLAY FETCH 'NAME' 'VECTOR' 'LENGTH'
.→--------------------------------------.
| .→--------. .→----------------------. |
| |0 0 0 0 0| | .→------. .→--. .→--. | |
| '~--------' | ↓ADAMS  | ↓PA1| ↓5 3| | |
|             | |BANKS  | |PS2| |5 3| | |
|             | |CROW   | |PP1| |4 3| | |
|             | |DEAN   | |PP2| |4 3| | |
|             | |EATON  | |PR1| |5 3| | |
|             | |FARR   | |PA1| |4 3| | |
|             | |GALVIN | |PS1| |6 3| | |
|             | |HARVEY | |PS1| |6 3| | |
|             | |INGRAM | |PA2| |6 3| | |
|             | |JACKSON| |   | |7 0| | |
|             | |KAHAN  | |PR2| |5 3| | |
|             | |LAMAR  | |PP1| |5 3| | |
|             | |MULVEY | |PS2| |6 3| | |
|             | '-------' '---' '~--' | |
|             '∊----------------------' |
'∊--------------------------------------'
```

Figure 15. Vector Format with Length Matrix

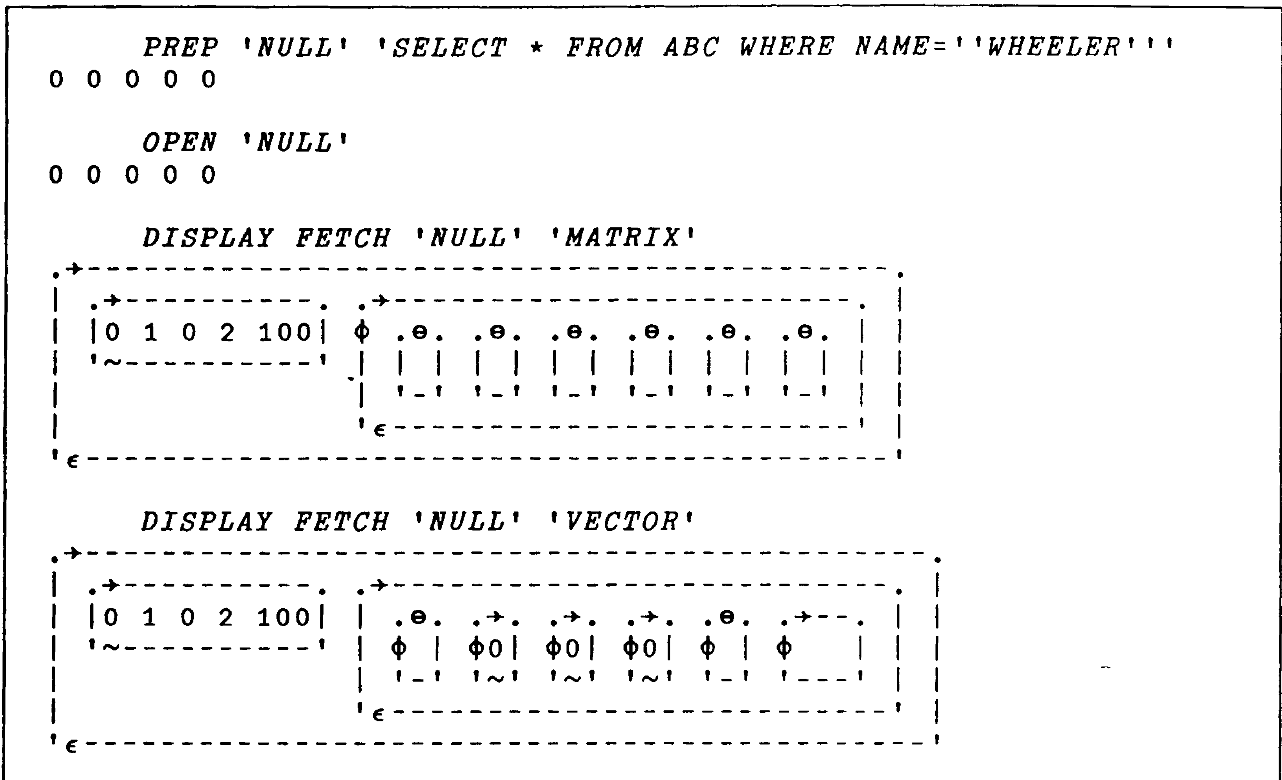
Figure 16 on page 16 shows the retrieval of a table in multiple steps, controlling the number of rows with the *FETCH*. In the case of such a small table, of course, one would not normally fetch the result in so many steps. The technique shown, however, may be used when space considerations prohibit fetching the entire table in one pass, or when the number of rows in the result cannot be predicted. The goal should be to minimize the number of calls to AP 127, given the current space constraints.

```
      PREP 'NAME' 'SELECT * FROM ABC'
0 0 0 0 0

      OPEN 'NAME'
0 0 0 0 0

      FETCH 'NAME' 5
0 0 1 0 0   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

      FETCH 'NAME' 5
0 0 1 0 0   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

      FETCH 'NAME' 5
0 0 0 0 0   KAHAN     6 3 32000 RES   PR2
            LAMAR    21 5 45000 PROD  PP1
            MULVEY    3 2 21000 SALES PS2

      CLOSE 'NAME'
0 0 0 0 0
```

Figure 16. Fetching a Table in Pieces

Note: After each *FETCH*, the cursor is left pointing at the last row fetched. AP 127 cannot tell that the end of the table is reached until it tries to fetch beyond the end. Therefore, if you fetch exactly the number of rows in the table, it is possible to get the (0 0 1 0 0) return code when the table is actually completely fetched. In that case, the next *FETCH* call will produce the return code (0 1 0 2 100), indicating there are no rows in the result.

If a result table has no rows, the data returned is an APL null prototype of the result. This lessens the need for special code to handle the null case, as the data will be the correct shape regardless of the number of rows fetched. Figure 17 on page 17 show what the null results look like.



```
      PREP 'NULL' 'SELECT * FROM ABC WHERE NAME=''WHEELER'''
0 0 0 0 0

      OPEN 'NULL'
0 0 0 0 0

      DISPLAY FETCH 'NULL' 'MATRIX'
.→-------------------------------------------.
| .→----------. .→------------------------.  |
| |0 1 0 2 100| ⌽ .⊖. .⊖. .⊖. .⊖. .⊖. .⊖. |  |
| '~----------' .| | | | | | | | | | | | | |  |
|               | '-' '-' '-' '-' '-' '-' |  |
|               'ϵ------------------------'  |
'ϵ-------------------------------------------'

      DISPLAY FETCH 'NULL' 'VECTOR'
.→---------------------------------------------.
| .→----------. .→--------------------------.  |
| |0 1 0 2 100| | .⊖. .→. .→. .→. .⊖. .→--. |  |
| '~----------' | ⌽ | ⌽0| ⌽0| ⌽0| ⌽ | ⌽   | |  |
|               | '-' '~' '~' '~' '-' '---' |  |
|               'ϵ--------------------------'  |
'ϵ---------------------------------------------'
```

Figure 17. Null Result Tables

Control Statements

LOCK, SET CURRENT SQLID

LOCK and **SET CURRENT SQLID** are both processed with the *EXEC* command, like data definition and authorization statements. No variables are allowed in these statements.

COMMIT, ROLLBACK

The **COMMIT** and **ROLLBACK** statements are processed by AP 127 commands of the same name. They cannot be processed with the *EXEC* command. The AP 127 commands allow an optional parameter, ***RELEASE***, which disconnects the user from the database. In SQL/DS, this corresponds to the **RELEASE** option. In DB2, it causes a disconnect from the Call Attach Facility, which AP 127 uses to communicate with DB2.

It is very important to be a responsible SQL citizen and issue COMMITs and ROLLBACKs as often as possible in your program. Not issuing them can cause data locks to be held, thus causing other users to wait for access to data. It is especially important to remember this in APL2, because implicit COMMITs are never issued, and implicit ROLLBACKs are issued only at shared variable retraction and *)OFF*. In compiled programming languages, the database connection is terminated when the program ends. In APL2, AP 127 is the program, so the connection stays active unless something explicit is done to terminate it.

Figure 18 on page 18 shows examples of *COMMIT* and *ROLLBACK* from APL2.

```
      ROLLBACK
0 0 0 0 0

      COMMIT
0 0 0 0 0

       DAT←'ROLLBACK' 'RELEASE'
       DAT
0 0 0 0 0
```

Figure 18. ROLLBACK, COMMIT

CONNECT

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, and optionally, the name of the database as well. The facility is useful, for example, when you want only one User ID to have certain authority or access to certain tables. An application can connect to that ID to do work in SQL, and individual users of the application need not have authority to use the data directly.

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 19 shows an example of the use of the AP 127 *CONNECT* command.

```
       CONNECT '' '' 'MYDATABASE'
0 0 0 0 0

       PREP 'NAME' 'SELECT * FROM INVENTORY'
1 0 0 2 ¯204

       PREP 'NAME' 'SELECT * FROM SQLDBA.INVENTORY'
0 0 0 0 0

       ROLLBACK
0 0 0 0 0

       CONNECT 'SQLDBA' 'SQLDBAPW' 'MYDATABASE'
0 0 0 0 0

       PREP 'NAME' 'SELECT * FROM INVENTORY'
0 0 0 0 0
```

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

Figure 19. CONNECT command

EXPLAIN

The **EXPLAIN** command, when issued from Dynamic SQL, is also processed with the ***EXEC*** command. Note that the statement is not placed in quotes. After the **EXPLAIN** is processed, you must look in the special SQL tables created for **EXPLAIN** to find the information. The structure and naming conventions for those tables are different in SQL/DS and DB2.

```
      EXEC 'EXPLAIN ALL FOR SELECT * FROM ABC'
0 0 0 0 0
```

Figure 20. EXPLAIN command

Error Handling

We have seen that the third item in the return code vector is used to flag the condition that there may be more rows in the result table.

The first and second items flag error or warning conditions. If the first is set to 1, there has been an error. If the second is set, there has been a warning. If both are set, an error has occurred which has caused a ***transaction backout***. This means that the current unit of work has been rolled back by the system.

The fourth and fifth items of the return code vector tell you the source of error or warning, and the error code.

The *MESSAGE* function may be used to retrieve more information about an error. It accepts as a parameter the return code vector from the AP 127 operation. The result from execution of the *MESSAGE* function depends on the type of error and the environment.

AP 127 Errors

Sometimes an error or warning is detected in AP 127 before the request is passed to SQL. When that happens, the fourth item in the return code vector is 1, and the fifth is the AP 127 message number. In Figure 21, 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 21. An AP 127 error

SQL Errors

Sometimes, although AP 127 detects no error, SQL cannot process the command. When an error or warning is discovered by SQL, the fourth return code is 2, and the fifth return code is the SQLCODE.

The *MESSAGE* function can also be used for SQL errors. It will return the contents of the SQLCA control block, which contains information about the status of the error in SQL. This information can be used in conjunction with the database message manual to debug the problem.

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. You can code another SQL query to retrieve the message text if you wish to.

Figure 22 shows what happens when we execute an SQL statement against a non-existent table.

```
      EXEC 'DELETE FROM BADTAB'
1 0 0 2 ¯204

      DISPLAY MESSAGE 1 0 0 2 ¯204
.→-----------------------.
↓ .→-----------------.   |
| |ERROR MESSAGE.    |   |
| '------------------'   |
| .→-----------------.   |
| ↓ .→---.           |   |
| | |¯204|           |   |
| | '~---'           |   |
| | .→-------------. |   |
| | |WHEELS  "BADTAB| |   |
| | '--------------' |   |
| | .→-------.       |   |
| | |ARIXOCA |       |   |
| | '--------'       |   |
| | .→-------------. |   |
| | |¯100 0 0 ¯1 0 0| |   |
| | '~-------------' |   |
| | .→-------.       |   |
| | |W   W   |       |   |
| | '--------'       |   |
| | .→-------.       |   |
| | |        |       |   |
| | '--------'       |   |
| '∊-----------------'   |
| .θ.                    |
| | |                    |
| '-'                    |
'∊-----------------------'
```

Figure 22. An SQL error

Warnings

Warnings do not always indicate a problem, but rather conditions which may warrant further inspection. AP 127, for example, issues a warning if your value-list is longer than necessary. A common error made in using value-lists is that a character vector is passed without enclosing it, and AP 127 cannot tell whether it is meant to be one value or several.

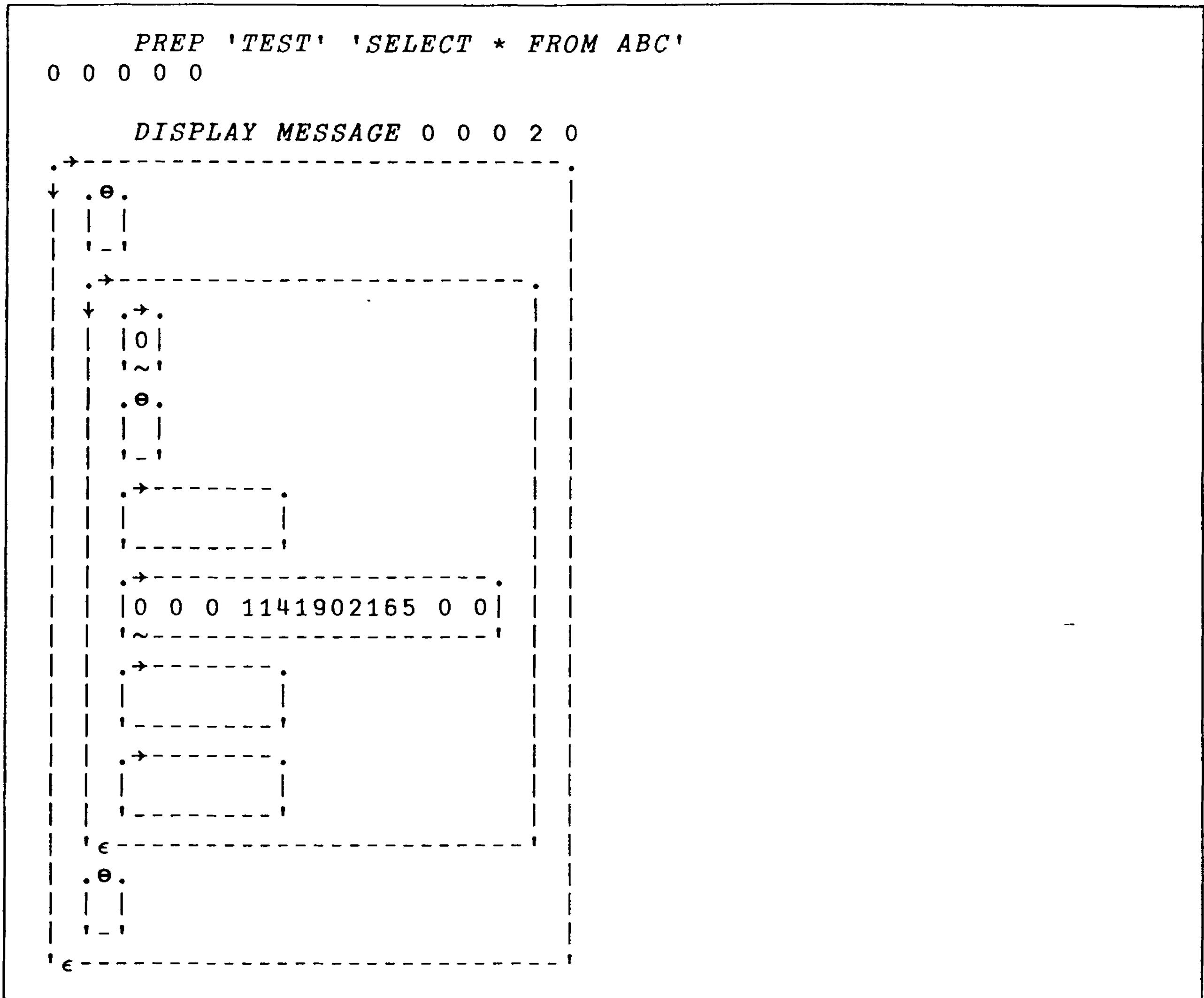
With SQL warnings, it is possible to receive a return code of the form (`0 1 0 2` *n*) or (`0 1 0 2 0`). In the first case, the error number tells you about the warning. In the second case, the warning flags are set in the SQLCA control block to indicate the type of warning.

In Figure 22 on page 20, you can see that in the second row from the bottom of the SQLCA, warning flags 0 and 4 are set to 'W'. The SQL documentation tells us that SQLWARN4 indicates that a **DELETE** statement has been issued without a **WHERE** clause. If the table BADTAB had existed, all rows would have been deleted.

Again, warnings may or may not indicate a real problem, but application code should be written to handle the possibility of them occurring.

Using the SQLCA

Even when no error has occurred, the contents of the SQLCA can sometimes be useful. For instance, after the **PREPARE**, a query cost estimate is placed in the SQLCA. You can examine the SQLCA anytime by passing the *MESSAGE* function a five-item numeric vector with a 2 in the fourth position. Figure 23 on page 22 shows what the SQLCA looks like after a successful **PREPARE**.



```
      PREP 'TEST' 'SELECT * FROM ABC'
0 0 0 0 0

      DISPLAY MESSAGE 0 0 0 2 0
.→---------------------------.
↓ .⊖.                        |
| | |                        |
| '-'                        |
| .→-----------------------. |
| ↓ .→.                    | |
| | |0|                    | |
| | '~'                    | |
| | .⊖.                    | |
| | | |                    | |
| | '-'                    | |
| | .→-------.             | |
| | |        |             | |
| | '--------'             | |
| | .→-------------------. | |
| | |0 0 0 1141902165 0 0| | |
| | '~-------------------' | |
| | .→-------.             | |
| | |        |             | |
| | '--------'             | |
| | .→-------.             | |
| | |        |             | |
| | '--------'             | |
| '∊-----------------------' |
| .⊖.                        |
| | |                        |
| '-'                        |
'∊---------------------------'
```

Figure 23. Query Cost Estimate

For a complete description of the SQLCA and information on the query cost estimate, see the database documentation.

Part 3: APL2 and SQL

Using APL2 and SQL together effectively is more than just knowing the mechanics of how to issue SQL statements from APL2. To write a successful application, one needs to consider such issues as database design, performance, storage usage, locking, and authority schemes as well as just "getting the right answers". We will try to address some of these issues here in light of the APL2/SQL interface, and we encourage those who are serious about using it to learn as much as possible about SQL and the particular implementation of it they will be using. Ignorance is not bliss, it is costly.

Locking

The types of locks obtained when accessing SQL tables, and the duration of those locks, can affect the perceived performance of an application. ***Concurrency*** is the term used to refer to the ability to have multiple users accessing data simultaneously. In general, with greater concurrency, there is less average waiting time for data access, and thus performance is perceived to be better.

The application can control some aspects of locking during execution, and others are determined at design time.

Lock Size

The size of the locks obtained is determined at the time the space for the tables is allocated. In DB2, a TABLESPACE can be locked in its entirety, or a page at a time. In SQL/DS, a public DPSPACE can be locked by DBSPACE, page, or row. A private DBSPACE is always locked in its entirety.

With TABLESPACE or DBSPACE locking, fewer locks are necessary but their scope is wider. Anyone else wishing to access the affected table or tables must wait until the current transaction is terminated before being granted that access.

With page locking, locks are obtained for the pages on which the rows accessed by the transaction reside. With row locking, locks are obtained only for the rows accessed by the transaction. If the transaction only accesses a few rows of the table, other users accessing different parts of the table may have a shorter wait. If the transaction accesses most of the rows of the table, however, the performance impact of obtaining many locks may be greater than that of the waits by other users. When deciding on a lock size, the types of transactions to be executed should be considered.

Isolation Level

The duration of locks is controlled by the ***Isolation Level*** attribute. There are two different isolation levels. With the ***Repeatable Read*** (RR) isolation level, locks are not released until the unit of work terminates with a COMMIT or ROLLBACK. With the ***Cursor Stability*** (CS) isolation level, locks are released when the cursor moves off the area (DBSPACE, TABLESPACE, page, or row) locked, if the data on the area has not been changed. If it has been changed, the locks are held until the unit of work terminates.

Repeatable read locking guarantees that if the same data is read twice during the same unit of work, the data will not have changed. Cursor stability locking allows more concurrency. Once an application is finished reading data and it moves on, other users can then access the data. However, CS also means that if the first

application reads the data again, it could be different than it was before. Table 2 on page 24 shows an example where data integrity could be lost.

Program Action	Database Action
P1 reads R1	R1 locked
P1 reads R2	R1 unlocked
P2 reads R1	R1 locked
P2 updates R1	
P2 commits	R1 unlocked
P1 updates R1, but R1 is different	R1 locked, but its integrity is uncertain

Table 2. Cursor Stability Locking

In general, applications that will do updates and depend on reading other data to determine which updates to make should not use CS locking. Applications that only read data can benefit from the added concurrency provided by CS locking.

In APL2, an AP 127 command, ***ISOL***, is provided that allows an application to change isolation levels during program execution. This allows the programmers to provide any desired mix of isolation levels in their applications.

In TSO, the new isolation level will take effect at the beginning of the next unit of work (after COMMIT or ROLLBACK). In CMS, the new isolation level is in effect for any cursors prepared after the change is made.

See Figure 24 for an example of the use of the ISOL command.

```
      ISOL ''        ⍝  Query the Isolation Level
0 0 0 0 0  RR

      ISOL 'CS'      ⍝  Set the Isolation Level
0 0 0 0 0
```

Figure 24. Forms of the ISOL command

Explicit Lock Control

In addition to controlling locking with installation parameters, locking may be explicitly controlled with the SQL **LOCK** command. The **LOCK** command will lock the entire TABLESPACE or DBSPACE in which the table being accessed resides.

Since the **LOCK** command causes the default locking to be overridden, another method for maximizing concurrency is to choose the CS isolation level with page or row locking, and use **LOCK** when doing updates to insure data integrity.

Authority

When using Dynamic SQL, authority must be granted to users to access the tables after they have been created. Different levels of authority can be granted (SELECT, UPDATE, INSERT, etc.); care should be taken that you do not grant more authority than is necessary to any user. There are many schemes for managing authority. We list a few here:

- Grant all authorities to all users. All users will then be able to access and update all data. This scheme is not advised unless the integrity of the data is unimportant.
- Grant SELECT authority to all users and UPDATE/INSERT authority to a limited number of users. This scheme offers more integrity but limits usability. All database update requests must be routed to one of the few users who can do them.
- Grant SELECT authority to all users for interactive data access and write batch programs using Static SQL to do updates. Programs written in VS FORTRAN and IBM 370 Assembler Language can be called from APL2 using the Name Association Facility. Authority to update in Static SQL is based on the authority to run the program rather than the authority to update the tables. This method restricts the updates as desired, but is less flexible since a new batch program must be written each time different type of update is desired.

  Note: In DB2, programs called from APL2 must be run with the DB2 Call Attach Facility, not the DSN command processor, since APL2 uses the Call Attach Facility.
- Define *views* for groups of users and allow the groups to update only their views. Views are logical tables whose definitions are based on real tables; users can access them much as they would a real table. If you want to use the views to update the data, however, you should read carefully about the ramifications of doing so. Because views do not physically exist, the update rules for them are somewhat more restrictive than the rules for tables.
- Use a central server user ID, and route all requests through the server. The server is, then, the only ID that has authority to access the data. The server's programs can be written to check each user's requests for correctness, and can further refine the authority scheme by allowing users to update only certain columns of the tables. See "Multi-User SQL Applications in APL2", Dr. James A. Brown (IBM TR 03.247) for a complete discussion of this scheme.
- (SQL/DS only) Create a special SQL user ID for your application, and use the CONNECT command in the application to connect the user as the application user ID. In this scheme, only the application user ID has authority to access the tables. All users will be able to make updates while running the application, but will not be able to access the tables outside the application from their own user ID. The updates, therefore, are restricted to the type and format allowed by the application.

The Index

Choosing appropriate indices for your tables is one of the most important design decisions you will make, because it can have a very great effect on the performance of your application. You can define indices on columns or combinations of columns, and indices can be unique (no duplicates are allowed in column(s) used to define the index) or nonunique.

If no indices are defined on a table, each row of the table must be searched every time you issue a query to find the rows that meet the conditions of the query. If there is an index on the column you specify in your search condition, the rows that meet the condition may be accessed directly, thereby speeding up the query.

Another effect of an index is that it can force uniqueness of data. If a UNIQUE INDEX is defined on a column or group of columns, the database will not allow duplicate data to be inserted into those columns.

The design question to be answered with indices is when to stop. How many indices is too many? If you define an index on every column in your table, you will have direct access to data whenever possible (there are some circumstances where an index cannot be used). The time required to update the table, however, will be increased. Every time a row is updated, each index must also be updated. The correct answer to the question is, of course, dependent on your application. Ideally, you will define indices on columns that will frequently be used in search conditions, and not define so many that updating the table is prohibitively slow.

Storage

Now that XA is universally available, and SQL tables are getting bigger and bigger, it may help to understand something of how AP 127 uses storage.

Shared Variable Storage

Because AP 127 is an auxiliary processor, it does not build its results directly into the workspace. It must allocate space out of free space for the shared variable contents. The buffer allocated can grow if large value-lists are passed to AP 127, or if large result tables are requested.

AP 127 does not set a restriction on the size of the buffer. Once the result is built, however, it is restricted by the SHRSIZE parameter and the workspace size, so it is possible to receive errors from AP127 or WSFULL conditions after AP 127 has successfully built the table.

The application can manage the shared variable buffer size in three ways. First, it can keep the number of rows fetched to a smaller size. Setting the AP 127 rows option to a very high number means that storage must be available for that number of rows, because AP 127 does not know in advance how many rows will actually be fetched. Second, it can use the *vector* format for the result table, which takes less space. Third, it can retract the shared variable after a particularly large fetch. On shared variable retract, AP127 will release the buffer if it has grown larger than the default size of 8K.

Statement Storage

Each statement that is prepared by AP127 has some storage allocated to it for status information, SQL control blocks and value-lists. For performance reasons, this storage is not freed every time the statement is called. Up to 40 statements may be active at one time, so this amount of storage may be significant in some cases.

The statement storage is freed at COMMIT and ROLLBACK time. If the application wants to clean up for itself, however, there is also an AP 127 command which will release storage for one or all statements. Figure 25 on page 27 shows the use of the *PURGE* command.

```
      PURGE 'NAME'        ⍝  Purges NAME only
0 0 0 0 0

      PURGE ''            ⍝  Purges all statements
0 0 0 0 0
```

Figure 25. PURGE command

Performance

There are numerous ways to tune APL2 and SQL applications. We have already discussed some of the issues involved in performance. They will be summarized here.

- Use the APL2 timing facility, available in APL2 Version 1 Release 3, to tune the APL portion of your application as much as possible.
- (SQL/DS Only) Try the BLOCK parameter on SQLPREP, if not already in use.
- Experiment with ISOL(CS) vs. ISOL(RR) in various scenarios. Caution: in a single-user environment, ISOL(RR) is faster, because less CPU time is used for it. To see the benefit of ISOL(CS), which is reduced **elapsed** time due to shorter waits, you must have multiple concurrent users.
- Vary the number of rows selected on each FETCH to minimize the number of AP 127 calls necessary to get the data.
- Make sure you have appropriate indices on your SQL tables.
- Optimize your queries. You can use the SQL EXPLAIN command to analyze query performance.
- Try the AP 127 ***vector*** form rather than the ***matrix*** form. In addition to saving space, it may also save you time.
- Do not repeat PREPARE requests unnecessarily. If you need to issue the same query with different parameters, PREPARE once with a placeholder and pass the fill-in value as you execute, rather than coding a literal string for the value in the SQL statement.
- Learn the AP 127 commands and use them directly rather than using the *SQL* function. You can often write more efficient code than the general purpose function allows for.
- If certain queries are repeated frequently and take a long time, consider coding these queries in a compiled language. The compiled program can be called separately or from APL2.
- (DB2 Only) In MVS, the asynchronous nature of the operating system allows the task calling DB2 to be separate from the main AP 127 task, and AP 127 is a separate task from APL2. If you use the shared variable system functions directly, you can do other work while waiting for the DB2 request to complete. You can also set a timer to limit the amount of time spent on a query. Retraction of the shared variable will cancel the request if it has not completed.
- Read more about SQL in the documentation for the database you are using. SQL is easy to use, and thus easy to use poorly. The more you know, the better your programs will be.

Appendix A. SQL Statement Summary

Statement Type	SQL Statement	AP 127 Processing Method
Data Definition	CREATE ALTER DROP LABEL ON COMMENT ON	EXEC
Data Manipulation	DELETE INSERT UPDATE	No Variables: EXEC With Variables: PREP,CALL
Query	SELECT	PREP,OPEN,FETCH,CLOSE
Authorization	GRANT REVOKE	EXEC
Control	LOCK SET CURRENT SQLID CONNECT COMMIT ROLLBACK	EXEC CONNECT COMMIT ROLLBACK
Analysis	EXPLAIN	EXEC

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 [type]	*'DESCRIBE' name [type]*
EVAL data	
EVALSIM data	
EXEC stmt	*'EXEC' stmt*
FETCH name [options]	*'FETCH' name [options]*
GETOPT	*'GETOPT'*
ISOL level	*'ISOL' [level]*
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]	
RESUME stack	
ROLLBACK	*'ROLLBACK'*
SETOPT options	*'SETOPT' options*
SHOW result	
SQL stmt [values]	
STATE name	*'STATE' name*

Function Name and Syntax	AP 127 Operation Code and Syntax
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.

Appendix D. Installation Instructions

Instructions are provided in the APL2 Installation Manuals for preparing SQL for access by APL2. However, since this preparation requires a deviation from the default install process, the necessary steps will be repeated here, with extra detail.

Note that these instructions are for APL2 Version 1 Release 3 only.

SQL/DS (CMS) Environment

To install the APL2 SQL interface:

1. LINK and ACCESS the minidisk in the SQL/DS machine. This is normally the 195 disk.

   ```
   LINK SQLDBA 195 195 RR
   ACC 195 Q
   ```

2. Initialize your user machine to the database by issuing the SQLINIT EXEC.

   ```
   SQLINIT DBNAME(SQLDBA)
   ```

   The SQLINIT process places onto your A-disk some files which are the SQL/DS "bootstrap" modules. These modules determine which SQL/DS database will be accessed. Once the files are created, there is no need to repeat this step unless they are erased or you wish to change to a different database.

3. (Optional) Modify the AP2V127I ASMSQL source module to add support for the SQL/DS Version 2 CONNECT extensions. Instructions for the modification are contained in the source module commentary.

   This step is necessary ONLY if you have SQL/DS Version 2 and VM/SP 5, and you wish to use multiple databases.

4. Preprocess the APL2 interface module. This step "registers" the APL2 program in SQL/DS.

   ```
   SQLPREP ASM PP(NOPUNCH,NOPRINT,PREP=AP2VSQL3),
           ISOL(USER),BLOCK,USER=SQLDBA/sqldbapw)
           IN(AP2V127I ASMSQL fm)
   ```

   or, if you have modified AP2V127I ASMSQL,

   ```
   SQLPREP ASM PP(NOPRINT,PREP=AP2VSQL3),
           ISOL(USER),BLOCK,USER=SQLDBA/sqldbapw)
           IN(AP2V127I ASMSQL fm)
           PU(AP2V127I ASSEMBLE fm)
   ```

   The SQLPREP command must be issued as shown, with the appropriate password and filemodes filled in. The one exception is the BLOCK parameter, which is optional. Since the use of BLOCK provides

significant performance improvements, it is highly recommended. Some SQL queries, however, do not work when BLOCK is in effect. For more information on BLOCK, see the SQL/DS Application Programmer's Guide.

5. Grant authority for users to use the APL2 interface.

```
CONNECT SQLDBA IDENTIFIED BY sqldbapw
GRANT RUN ON AP2VSQL3 TO PUBLIC
```

   The CONNECT and GRANT commands can be issued using the ISQL processor, or with the SQLDBSU EXEC.

6. (Optional) If you have modified AP2V127I ASMSQL, and created AP2V127I ASSEMBLE, assemble the module. Assembler H is required. The AP2MAC MACLIB must be available, and a GLOBAL MACLIB AP2MAC statement issued before running the assembly.

   Place the resulting AP2V127I TEXT file on a disk that precedes the APL2 installation disk in the CMS search order, and generate the APL2 load module according to the instructions provided in the CMS installation guide. The installation process will replace the default AP2V127I TEXT with your modified one.

If the installation has multiple SQL/DS databases, and APL2 will be used from each of them, steps 1 through 5 must be repeated for each database.

To use the APL2 SQL interface:

1. See your database administrator to be granted appropriate authority to access the database and obtain space for creating tables. There are different levels of authority. For more information, see the SQL/DS Planning and Administration Guide.
2. LINK and ACCESS the SQL/DS minidisk, as in step one above.

   If you will be using SQL and APL2 together frequently, you may want to add the LINK and ACCESS to your AP2EXIT EXEC so it is done automatically each time you invoke APL2.
3. Issue SQLINIT as in step 2 above, if the bootstrap modules do not already exist on your A-disk.
4. Invoke APL2

DB2 (TSO) Environment

To install the APL2 SQL interface:

After the SMP APPLY step for the APL2 product, run the AP2JBIND job. This will bind and authorize users to run the APL2 application plans.

If the installation has multiple DB2 subsystems, and APL2 will be used from each of them, the above steps must be repeated for each subsystem.

To use the APL2 SQL interface:

1. See your database administrator to be granted appropriate authority to access the database and obtain space for creating tables. There are different levels of authority. For more information, see the DB2 Reference Manual.

2. ALLOCATE the DB2 load libraries in your TSO LOGON PROC or your APL2 CLIST.
3. Invoke APL2

Appendix E. References

DB2 Publications

1. IBM Database 2 SQL User's Guide (SC26-4376)
2. IBM Database 2 Application Programming Guide (SC26-4377)
3. IBM Database 2 SQL Reference (SC26-4380)
4. IBM Database 2 Reference Summary (SX26-3771)
5. IBM Database 2 Messages and Codes (SC26-4379)

SQL/DS Publications

1. SQL/DS SQL Reference (SH09-8069)
2. SQL/DS Database Planning and Administration (SH09-8017)
3. SQL/DS System Planning and Administration (SH09-8018)
4. SQL/DS Application Programming Guide (SH09-8019)
5. SQL/DS Messages and Codes (SH09-8052)

APL2 Publications

1. APL2 Programming: Using Structured Query Language (SH20-9217)
2. APL2 Programming: Guide (SH20-9216)
3. APL2 Programming: System Services Reference (SH20-9218)
4. APL2 Reference Summary (SX26-3737)

Other

1. Development Guide For Relational Applications (SC26-4130)
2. Techniques in SQL Application Design (TR 03.290)
3. Interactive SQL and APL2 (TR 03.289)
4. SQL News You Can Use (SEAS SM88)