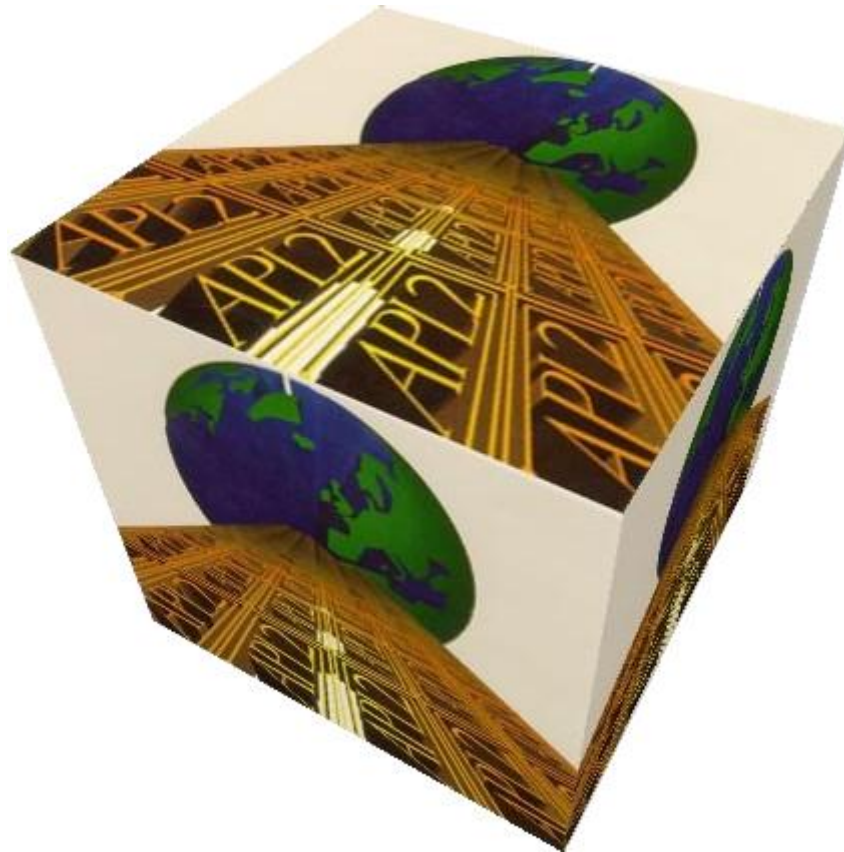


APL2 Programming: Using SQL

SH21-1057-07

**APL Products and Services
IBM Silicon Valley Laboratory
555 Bailey Avenue
San Jose, California 95141
APL2@vnet.ibm.com**



Copyrights

© Copyright IBM Corporation 1984, 2008 All Rights Reserved.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corporation

Contents

- [Notices](#)
- [We Would Like to Hear from You](#)
- [Preface](#)
- [Overview](#)
- [Auxiliary Processor Communications](#)
- [Auxiliary Processor Operations - Reference](#)
- [The SQL Workspace](#)
- [The SQL Workspace - Reference](#)
- [Special Programming Techniques](#)
 - [Concurrent Update of Shared Tables](#)
 - [Interrupting SQL Processing](#)
 - [Handling the LOB Datatypes](#)
 - [Support for DBCS Characters](#)
- [APL2/SQL Interface Messages](#)
- [Quick Reference Tables](#)
 - [Auxiliary Processor Operation Codes](#)
 - [SQL Workspace Functions](#)
 - [Return Code Vector](#)
 - [SQL Statement Types and APL2 Operations](#)
 - [SQL and APL2 Data Types](#)
- [Glossary](#)

Notices

References in this publication to [IBM](#) products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe on any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject material in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Programming Interface Information

This book is intended to help you use [APL2](#) with the Structured Query Language (SQL) to access relational database systems. It documents *General-Use Programming Interface and Associated Guidance Information* provided by APL2. General-use programming interfaces allow the customer to write programs that obtain the services of APL2.

Trademarks

IBM Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX/6000

APL2

DATABASE 2

DB2

GDDM

IBM

OS/2

SQL/DS

System/370

System/390

Other Trademarks

The following terms are trademarks of other companies:

Sun	Sun Microsystems, Inc.
Solaris	Sun Microsystems, Inc.
Windows	Microsoft Corporation

We Would Like to Hear from You

APL2 Programming: Using SQL

Please let us know how you feel about this online documentation by placing a check mark in one of the columns following each question below:

To return this form, print it, write your comments, and mail it to:

International Business Machines Corporation
APL Products and Services - H36A/F40
555 Bailey Avenue
San Jose, California 95141
USA

For postage-paid mailing, please give the form to your IBM representative.

You can also send us your comments by email. To send us this form, copy it to a file, write your comments using a file editor, and then send it to:

apl2@vnet.ibm.com

Overall, how satisfied are you with the online documentation?

	Very Satisfied			Very Dissatisfied
	1	2	3	4
Overall Satisfaction	---	---	---	---

Are you satisfied that the online documentation is:

Accurate	---	---	---	---
Complete	---	---	---	---
Easy to find	---	---	---	---
Easy to understand	---	---	---	---
Well organized	---	---	---	---
Applicable to your tasks	---	---	---	---

Please tell us how we can improve the online documentation:

Thank you! May we contact you to discuss your responses?

___Yes ___No
Name: _____

Title: _____

Company or Organization:-----

Address:-----

Phone:-----
(____)-----
E-mail:-----

Please do not use this form to request IBM publications. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office servicing your locality.

Preface

This book is intended to help you use [APL2](#) with the Structured Query Language (SQL) to access relational database systems.

It is assumed that you:

- Understand APL2 syntax, data structures, the structural and selection functions, and the *each* (**) operator
- Know how to load workspaces with the) LOAD system command and how to use defined functions and operators
- Understand the basic concepts of shared variables and the general uses of auxiliary processors (APs)
- Are familiar with the concepts of relational databases and the facilities of the Structured Query Language

The APL2 primitive functions used in the illustrations in this book include *depth* ($\equiv A$), *enclose* ($\subset A$), *first* ($\uparrow A$), *pick* ($\mathbb{I} \supset A$), and *specification* ($L \leftarrow R$). These and all other primitives are described in *APL2 Programming: Language Reference*.

In all examples, index origin 1 ($\square \mathbb{I} O \leftarrow 1$) is assumed.

Conventions Used in This Manual

- lower* Lowercase italicized words in syntax represent values you must provide.
- UPPER In syntax blocks, uppercase words in an APL character set represent keywords that you must enter exactly as shown.
- [] Usually, brackets are used to delimit optional portions of syntax; however, where APL2 function editor commands or fragments of code are shown, brackets are part of the syntax.
- [A | B | C] A list of options separated by | and enclosed in brackets indicate that you can select one of the listed options.
Here, for example, you could specify either A, B, C, or none of the options.
- {A | B | C} Braces enclose a list of options (separated by |), one of which you must select.
Here, for example, you would specify either A, B, or C.
- ... An ellipsis indicates that the preceding syntactic item can be repeated.
- { }... An ellipsis following syntax that is enclosed in braces indicates that the enclosed syntactic item can be repeated.

This manual refers to APL2-defined functions and auxiliary processor operation codes that have the same name. `FETCH`, for instance, is both a defined function and an auxiliary processor operation code.

Because the defined functions and the equivalent auxiliary processor operations perform the same tasks, you can use either in many processing situations. When the distinction between the two is not important, the manual lists the name without putting it in the APL font (slanted alphabetic capitals) followed by the word *request*, rather than *operation* or *function*. Thus:

`FETCH` request

Refers to both the defined function and the auxiliary processor operation.

`FETCH` function

Refers only to the defined function.

'`FETCH`' operation

Refers only to the auxiliary processor operation.

The term *workstation* refers to all platforms where APL2 is implemented except those based on [System/370](#) and [System/390](#) architecture.

The term *mainframe* refers to platforms based on System/370 and System/390 architecture.

APL2 Documentation

Along with this manual, *APL2 Programming: Using SQL*, the following additional on-line manuals are included with Workstation APL2:

- *APL2 User's Guide*
- *APL2 Language Summary*
- *APL2 Programming: Language Reference*
- *APL2 Programming: Developing GUI Applications* (for Windows only)
- *APL2 Programming: Using APL2 with WebSphere*
- *APL2 GRAPHPAK: User's Guide and Reference*

The following table shows all the books in the APL2 library, organized by the tasks for which they are used. The APL2 library can be found on the web at <http://www.ibm.com/software/awdtools/apl/library.html>.

Information	Book	Publication Number
Introductory language material	An Introduction to APL2 APL2 Language Summary	SH21-1073 SX26-3851
Common reference material	APL2 Programming: Language Reference APL2 Reference Summary APL2 GRAPHPAK: User's Guide and Reference APL2 Programming: Using SQL APL2 Migration Guide	SH21-1061 SX26-3999 SH21-1074 SH21-1057 SH21-1069
Mainframe programming	APL2 Programming: System Services Reference APL2 Programming: Using the Supplied Routines APL2 Programming: Processor Interface Reference APL2 Installation and Customization under CMS APL2 Installation and Customization under TSO APL2 Messages and Codes APL2 Diagnosis	SH21-1054 SH21-1056 SH21-1058 SH21-1062 SH21-1055 SH21-1059 LY27-9601
Workstation programming	APL2 User's Guide APL2 Programming: Developing GUI Applications APL2 Programming: Using APL2 with WebSphere	SC18-7021 SC18-7383 SC18-9442

Overview

Through the SQL auxiliary processor, AP 127, APL2 provides access to IBM relational database systems DB2 and SQL/DS. Through the ODBC auxiliary processor, AP 227, APL2 provides access to databases and other programs that support the ODBC protocol. Except where noted, the same SQL language is used with both AP 127 and AP 227, and both support the same auxiliary processor commands and results.

SQL is a high-level language that uses the relational data model. A *relation* in the relational data model can be thought of as a simple two-dimensional table - a matrix in APL2 terms. The columns of the table have names, and the rows contain data. The intersection of a row and column is called a *value* or a *field*. Each column can be defined for either numeric or character data, and each field in the table can contain data or it can be empty.

SQL users need not know how data is represented in storage in order to retrieve and use it. Using SQL statements, the database management system finds its own way to the data. As an SQL user, you access the data you want by specifying what data you want rather than how to access it. The database management system schedules the file read/write sequences and manages data integrity so that you are concerned with the data you want to store, retrieve, or manipulate and not with scheduling the file input/output.

Note: The database system is usually managed by a database administrator, who authorizes certain kinds of access. The discussion of using SQL under APL2 assumes you are authorized for database access.

Structure of an SQL Table

Each SQL table is identified by a table name; each column within the table is given a column name. Here is a sample SQL table called STOCKS.

```
Table: STOCKS
BIN YEAR TYPE          STORLOC          COST COLOR
--- ---- -
B10 1973 CHAMBERTIN    COUNTER          10.95 R
B11 1966 BORDEAUX     SPECIALTY CORNER 8.95 R
B12 1974 BORDEAUX     SPECIALTY CORNER 10.95 R
C11 1971 RIOJA        IMPORT DEPARTMENT 4.95 R
C12 1971 RIOJA        WINE CLUB SHOWCASE 3.95 R
F16 1983 ROSE         WINDOW           NULL NULL
G12 1974 MERLOT       BASEMENT         8.95 R
I10 1979 BARDOLINO     ANNEX            2.25 R
I11 1979 VALPOLICELLA ANNEX            2.25 R
K10 1981 CHABLIS      ON ORDER         3.95 W
K11 1981 RIESLING     SHELF            6.25 W
```

Note: An actual SQL table is likely to contain many more rows.

When such a table exists, an SQL SELECT statement can retrieve data from the table by specifying what data is wanted; the specifications are called *search conditions* and are contained in a WHERE clause. For example, the following statement selects all wines that cost less than \$6.00 and whose color is 'R'. The asterisk (*) indicates that all columns of the table are to be returned.

```
SELECT * FROM STOCKS WHERE COST < 6.00 AND COLOR = 'R'
```

The resulting table is:

```
BIN YEAR TYPE          STORLOC          COST COLOR
--- ---- -
C11 1971 RIOJA        IMPORT DEPARTMENT 4.95 R
C12 197 RIOJA        WINE CLUB SHOWCASE 3.95 R
I10 1979 BARDOLINO     ANNEX            2.25 R
I11 1979 VALPOLICELLA ANNEX            2.25 R
```

Structured Query Language

SQL is a nonprocedural language with a straightforward grammar. SQL statements refer to SQL tables and column names known to the database system.

SQL statements may be categorized according to the type of function performed. The following table lists eight types of statements, their corresponding SQL keywords, and their effect on the database system.

Statement Type	SQL Statements	Effect
Query	SELECT	<p>Describes a <i>result table</i> as a function of a set of existing tables. For example:</p> <pre>SELECT NAME, SALARY FROM PAYROLL WHERE SALARY < 300.00</pre> <p>The rows of a result table are accessed one at a time with a <i>cursor</i> that points to the current row of the result table.</p>
Data Manipulation	DELETE INSERT UPDATE	<p>Deletes rows from an existing table.</p> <p>Inserts rows into an existing table.</p> <p>Changes the values of fields in a table.</p>
Data Definition	CREATE ALTER DROP ACQUIRE COMMENT ON LABEL ON	<p>Defines a table, tablespace, view, index, synonym, database, storage group, or DBSPACE.</p> <p>Changes the description of a table, tablespace, index, storage group, or DBSPACE.</p> <p>Removes a table, tablespace, view, index storage group, or DBSPACE and its description from the catalog.</p> <p>(CMS only) Gets a DBSPACE in which tables and indexes can be created.</p> <p>Places a comment on a table, view, or column.</p> <p>Places a label on a table, view, or column.</p>
Authorization	GRANT REVOKE	Controls access to data and privileges on the database system.
Control	LOCK TABLE COMMIT ROLLBACK CONNECT	<p>Locks a table.</p> <p>Permits explicit control of the disposition of a unit of work. Both statements remove locks from tables.</p> <p>Specifies the database to be accessed, user id, and connection options.</p>

Statement Type	SQL Statements	Effect
Procedure Call	CALL EXECUTE	Transfers control to a stored procedure.
Analysis	EXPLAIN	Provides information about the execution of a query for performance analysis.

APL2/SQL Interface

The APL2/SQL interface consists of:

- Auxiliary processors (AP 127 and AP 227) which accept requests by shared variables, transform them into standard runtime SQL requests, and pass them on to the database system.

Use of AP 127 and AP 227 is characterized by operation codes based on dynamic SQL, straightforward protocol, and simple, consistent syntax.

- The APL2 workspace named SQL, which provides programmer aids for communicating with the auxiliary processor. The functions in the workspace allow command stacking with error recovery.

Processing SQL Statements under APL2

The SQL language supported by AP 127 and AP 227 is a subset of that provided by for PL/I, C, COBOL, FORTRAN, and assembler language programmers.

On mainframe systems, AP 127 runs as a database application program with dynamic SQL statements embedded. On workstation systems, AP 127 makes calls to DB2 using the DB2 CLI driver. AP 127's operation codes are based on the statements used in dynamic SQL. Explicit cursor control is allowed through PREP-OPEN-FETCH-CLOSE sequences.

AP 227 makes calls to the ODBC Driver Manager using a set of routines provided by the Driver Manager. It uses the same operation codes as AP 127, providing the same level of functionality, including cursor control.

AP 227 uses application programming interfaces (API's) from the ODBC 3.0 standard, so the ODBC Driver Manager being used must support the ODBC 3.0 level or later.

SQL statements to be executed from APL2 can be described by the sequence of requests (auxiliary processor operations or SQL workspace functions) necessary to complete their processing.

The following table describes the APL2 commands that provide access to SQL tables. Each of the commands is available as a defined access function in the SQL workspace or as a direct auxiliary processor operation.

The complete description of the syntax of the access operations, and all other operations permitted, is contained in [Auxiliary Processor Operations - Reference](#). The workspace functions are described in [SQL Workspace Reference](#).

AP Command	Purpose
CALL	Invokes a data manipulation statement or procedure call processed by a previous PREP request and, if appropriate, specifies an APL2 value list as an additional parameter. The value list is indexed using the references in the statement processed by the PREP request. Valid only for nonquery statements. Equivalent to an EXECUTE in dynamic SQL. Equivalent to SQLBindParameter followed by SQLExecute in ODBC.
CLOSE	Closes an open cursor. Valid only for a statement already processed by an OPEN request.

AP Command	Purpose
COMMIT	Makes database changes since last COMMIT or ROLLBACK permanent.
CONNECT	Specifies the database to be accessed, user id, and connection options.
DECLARE	Declares a name to be used to refer to an SQL statement, and assigns attributes to the name.
DESCRIBE	Provides column names and data types for a prepared SELECT statement or stored procedure result set. This is analogous to an SQL DESCRIBE but does not always cause one to occur.
EXEC	Executes an SQL statement with no variable arguments. Any values for the statement must be embedded in the statement. Not valid for query (SELECT) statements or stored procedure calls with output parameters. Equivalent to an EXECUTE IMMEDIATE in dynamic SQL. Equivalent to SQLExecDirect in ODBC.
FETCH	Reads a specified number of rows of a result table into a shared variable. Valid only for a query statement already processed by an OPEN request.
OPEN	<p>Sets a cursor and, if appropriate, specifies an APL2 value list as an additional parameter.</p> <p>For SELECT statements, the OPEN request creates a result table that can then be retrieved with a FETCH request. For INSERT statements (CMS Only), the OPEN request readies the internal buffer to receive data from the PUT request. For called stored procedures, the OPEN request accesses the next available result set for retrieval with a FETCH request.</p> <p>A value list may be passed to the OPEN request only for SELECT statements. The value list is indexed using the references in the statement processed by the PREP request.</p>
PREP	<p>Prepares a statement for a subsequent CALL or OPEN request. The statement can contain references (indexes) to an APL2 array (value list) that are specified in a subsequent CALL, OPEN, or PUT request.</p> <p>There can be up to 40 prepared statements at any time.</p>
PUT (CMS only)	Moves the data into the internal buffer. When the buffer is full or the cursor is closed, the data is transferred to the database automatically. Valid for INSERT statements only. Requires a value list.
ROLLBACK	Cancels database changes since last COMMIT or ROLLBACK.

The next table summarizes the APL2 processing method that can be used to execute the various SQL statement types.

SQL Statement Type	SQL Statement	Processing Method
Query	SELECT	<p>The following sequence only:</p> <p>DECLARE (optional), PREP, OPEN, FETCH, CLOSE</p> <p>Values for SQL clauses are embedded or passed as a vector on the OPEN request.</p>
Data	DELETE	Either:

SQL Statement Type	SQL Statement	Processing Method
Manipulation	INSERT UPDATE	EXEC (values must be embedded in SQL statement) or: PREP, CALL (values are either embedded or passed as APL2 data on the CALL request) In CMS only, the PREP, OPEN, PUT, CLOSE sequence can also be used for INSERT.
Data Definition	CREATE ALTER DROP ACQUIRE COMMENT ON LABEL ON	EXEC request
Authorization	GRANT REVOKE	EXEC request
Control	LOCK TABLE COMMIT ROLLBACK CONNECT	EXEC request COMMIT, ROLLBACK, and CONNECT are submitted directly to the database management system. They are not only SQL statements, but also APL2/SQL interface requests.
Procedure Call	CALL EXECUTE	PREP, CALL Values for parameters declared in the procedure as OUTPUT or INOUT must be passed as APL2 data on the CALL request. In general, values for parameters declared in the procedure as INPUT may be either embedded in the statement or passed as APL2 data on the CALL request. On some database systems, however, values for parameters declared in the procedure as INPUT must be embedded. Then, for each result set built by the procedure: OPEN, FETCH Then, to free all result sets: CLOSE
Analysis	EXPLAIN	EXEC request

Value Substitution under APL2

Most SQL statements can contain values embedded in the statement. For example, the following SELECT statement contains the values for search conditions embedded in the statement itself.

```
SELECT NAME FROM PAYROLL-TABLE  
WHERE DEPT = 'M75' AND SALARY < 45000
```

The values 'M75' and 45000 are the search conditions for the SELECT statement.

SQL statements processed with an EXEC request must contain any values as part of the statement.

Query and data manipulation (SELECT, INSERT, UPDATE, DELETE) statements may use placeholders for values. Procedure calls must use placeholders for the procedure's OUTPUT and INOUT parameters, if any. These statement types (and only these) can contain references to an APL2 vector or matrix of values - the *value list*. Each reference is a *vector index* preceded by a required colon.

The vector indexes are passed in a PREP request, and the value list is passed in a subsequent CALL, PUT, or OPEN request. For example, given the following cursor statement, the :1 in the SELECT statement is replaced with the first item of a vector of values passed as an argument of an OPEN request. Similarly, the :2 is replaced with the second item of the value list.

```
SELECT NAME FROM PAYROLL-TABLE  
WHERE DEPT = :1 AND SALARY < :2
```

For a SELECT statement in this form, a value list must be included on the subsequent OPEN request for the statement. The value list for this example would contain 'M75' 45000 as its first two items because the vector indexes point to those two items.

The following example shows a call to a stored procedure with three arguments:

```
CALL PAYPROC(:1, :2, :3)
```

A value list can contain more items than are actually indexed, and the vector indexes can select items in any order. If the value list contains more items than the highest vector index, a warning message is issued.

If a value list is a matrix, and the statement is INSERT, UPDATE, or DELETE, the rows of the matrix are processed one at a time. SQL is called to execute the statement for each row of the matrix, with different values substituted for the vector indexes on each call. For SELECT statements, the first row of the matrix is used, and the remaining rows are ignored. For stored procedure calls, matrix value lists are not accepted.

When passing value list data to SQL, the SQL datatype that most closely corresponds to the data found in the value list is used. If that datatype does not match the type of the target column, SQL will attempt to convert the data to the target type. If the data cannot be converted to the target type within the rules established by the database system, an SQL error will result.

Because APL2 does not have the concept of binary character data, special syntax must be used by the APL2 programmer if the target column's type is BLOB (Binary Large Object), BINARY, or character declared with the FOR BIT DATA option. In the following example, the second column is a character column with the FOR BIT DATA attribute, and the fourth column is a BLOB column:

```
INSERT INTO BINTABLE VALUES (:1, :B2, :3, :B4)
```

Where the letter B immediately follows the colon, the auxiliary processor will use a binary character type to pass the corresponding value to the database.

Note: AP 127 and AP 227 are not sensitive to \square IO. Thus :0 is never valid as a vector index.

Rules for Coding SQL Statements in APL2

The SQL statements used with APL2 must conform to the syntax rules for the SQL language. There are some additional requirements for SQL statements as used in the APL2 environment. These requirements (and some clarification) are listed below for the four types of SQL statements, classified by their interaction with APL2.

1. Statements submitted by an EXEC request
 - Cannot contain any vector index references (equivalent to host-variable names) to APL2 value lists.
 - Cannot be SELECT (query) statements.
2. Data Manipulation Statements
 - Begin with one of three keywords - INSERT, UPDATE, or DELETE.
 - Can use vector indexes in search conditions if processed by a PREP request. Each index in the statement specifies an item in an array of values passed as an argument of a subsequent CALL or PUT request. Each index reference must be prefixed by a colon.
 - Can use CURRENT OF *cursorname* phrases only if they refer to a query (SELECT) statement that has been successfully processed with a DECLARE or PREPARE request, and contains the FOR UPDATE clause.
 - Can use CURRENT OF *cursorname* phrases with AP 227 only if the ODBC driver for the database system supports the SQLSetCursorName API, which allows AP 227 to associate cursor names with SQL statements.
 - Do not use indicator variable names.
3. Query (Cursor) Statements
 - Begin with the keyword SELECT. In the newest levels of the SQL language, a WITH clause may optionally precede the actual SELECT statement.
 - Must not have an INTO clause. All results are placed in a shared variable.
 - Can use vector indexes (in place of host-system variables) in search conditions. Each index in the statement specifies an item in an array of values passed as an argument of a subsequent OPEN request. Each index reference must be prefixed by a colon.
 - Do not use indicator variable names.
4. ROLLBACK/COMMIT
 - ROLLBACK WORK (in CMS, ROLLBACK WORK RELEASE) is automatically issued when the shared variable is retracted or when the APL2 session ends.
 - AP 127 and AP 227 do not issue any automatic COMMIT. On workstation systems, some databases support a built-in autocommit facility. Where this is supported, it can be requested when the database connection is made. See [CONNECT](#) for more information.
5. Procedure Calls
 - Begin with the keyword CALL, or on some database systems, EXECUTE.
 - Must use placeholders and value lists for the procedure's parameters that are declared as OUTPUT or INOUT, if any. These parameters may be updated by the procedure. The updated value list will be returned as part of the result of the CALL request. For parameters that are

updated by the procedure, if the size of the value list item for the parameter is too small to contain the updated value, the output data will be truncated.

- May build result sets, in the form of tables. The result sets are accessed using the OPEN, FETCH sequence of AP requests, after the CALL request has been completed. Each OPEN request will access the next available result set. When no more result sets are available, a warning return code will be returned by the OPEN request. On workstation systems, there is no limit on the number of result sets that may be fetched. On mainframe systems, up to 10 result sets may be fetched. Issuing a CLOSE request will terminate access to all result sets built by the procedure.
- Procedure parameters that are structures or arrays are not supported by AP 127 and AP 227.

Ways to Use SQL under APL2

You can communicate with AP 127 and AP 227 through the following methods:

1. By using the user support functions in the SQL workspace
2. By using the access functions in the SQL workspace
3. Directly, through shared variable operations
4. By using support functions that you write, which can in turn use one or more of the methods above

Method 1 requires the least work from you, but allows the least control and customization because auxiliary processor command sequences are generated automatically. You can concentrate on the SQL statements and data and allow the user support functions to handle the auxiliary processor requests.

Method 2 couples ease of access with specific sequences of operations. It allows you to control the sequence of auxiliary processor commands, but the shared variable communications are still automatic.

Method 3 is preferable if you write your own support functions. You handle the shared variable access yourself.

Method 4 can be used to tailor usage to frequently-used access sequences or to take advantage of full-screen facilities. For such applications, the functions in the SQL workspace can serve as both tools and guides. This method allows for the maximum control, efficiency, and customization usually needed by production applications.

A Beginning Example

With the user support functions provided in the SQL workspace, you can assign SQL statements to APL2 variables, use the defined function named SQL to create and pass the appropriate requests to the auxiliary processor, and return the results to the workspace.

The following example shows how to use the SQL function to create the STOCKS table.

Assume a variable CWINE that has the following value:

```
CWINE
CREATE TABLE STOCKS
  (BIN CHAR(3) NOT NULL,
  YEAR SMALLINT,
  TYPE VARCHAR(12) NOT NULL,
  STORLOC VARCHAR(20),
  COST DECIMAL(6,2),
  COLOR CHAR(1))
```

The CREATE TABLE statement defines the data structure of the table. The column-names - BIN, YEAR, and so on - are each associated with an SQL data type - CHAR, SMALLINT, and so on. (For a full description of the CREATE TABLE statement and SQL data types, see the *SQL Reference* for your database system.

The next variable, IWINE, is an INSERT statement that places data values into the table.

The VALUES portion of the statement contains vector indexes that index into another variable, WVALS, that contains the corresponding values.

You can use the EVAL function to make creating WVALS easier. See [EVAL](#) for more information.

```

      IWINE
INSERT INTO STOCKS
VALUES (:1, :2, :3, :4, :, :6)
      WVALS
B10 1973 CHAMBERTIN    COUNTER      10.95 R
B11 1966 BORDEAUX     SPECIALTY CORNER  8.95 R
B12 1974 BORDEAUX     SPECIALTY CORNER  10.95 R
C11 1971 RIOJA        IMPORT DEPARTMENT  4.95 R
C12 1971 RIOJA        WINE CLUB SHOWCASE 3.95 R
F16 1983 ROSE         WINDOW
G12 1974 MERLOT       BASEMENT      8.95 R
I10 1979 BARDOLINO    ANNEX        2.25 R
I11 1979 VALPOLICELLA ANNEX        2.25 R
K10 1981 CHABLIS      ON ORDER     3.95 W
K11 1981 RIESLING     SHELF        6.25 W

```

Note: The SQL NULL fields are represented as empty numeric or character vectors in the APL2 workspace.

The function SQL processes the INSERT statement contained in the variable IWINE, inserting the values from the variable WVALS into the STOCKS table. For each statement processed, it presents a five-item return code vector (all zeros indicates successful completion).

```

      CONNECT 'WINEDATA' 'MYUSERID' 'MYPASSWD'
0 0 0 0 0      SQL02011 WINEDATA MYUSERID
      SQL CWINE
0 0 0 0 0
      SQL IWINE WVALS
0 0 0 0 0
      COMMIT
0 0 0 0 0

```

Note: The last statement, COMMIT, ensures that the updates to the database are made permanent.

The SQL function handles all the necessary shared variable communications with the auxiliary processor. You need to be concerned only with the SQL statements and their results.

[The SQL Workspace](#) contains a general description of the workspace; the syntax of each defined function and the defined operator is described in [The SQL Workspace - Reference](#).

After the table is created and values have been inserted into its rows, you can retrieve data from it as with the following expression used with the SQL workspace functions.

```

SQUERY←'SELECT * FROM STOCKS WHERE COST < 6.00 AND COLOR = ''R'''
(RC DATA STACK)←SQL SQUERY

```

After execution of the request, the SQL function returns a three-item vector. The first item is the return code; the second is a table of red wines that cost less than \$6.00. The third item is the stack of unexecuted operations. The stack is empty if the commands were processed without error. The table can be returned as a matrix, or as a vector containing a simple matrix for each column.

```
      RC
0 0 0 0 0
DATA
C11 1971 RIOJA      IMPORT DEPARTMENT 4.95 R
C12 1971 RIOJA      WINE CLUB SHOWCASE 3.95 R
I10 1979 BARDOLINO  ANNEX              2.25 R
I11 1979 VALPOLICELLA ANNEX            2.25 R
```

This data can now be processed by the APL2 application.

Summary

This chapter has introduced some basic concepts of SQL, and shown a simple, high-level example of accessing SQL from APL2. With this information, you can begin using the two products together.

To use SQL most successfully from APL2, however, requires more in-depth knowledge of the SQL language and the APL2 auxiliary processor used with it. The rest of this book gives details about using AP 127 and AP 227. directly or by the supplied workspace SQL.

For more information on the SQL language and the associated relational database products, consult the libraries of those products.

Auxiliary Processor Communication

This section contains the following topics:

- [Shared Variable Requirements](#)
- [Arguments](#)
- [Results](#)
- [Operation Dependencies](#)
- [Terminating the Unit of Work](#)

Shared Variable Requirements

AP 127 and AP 227 communication with the APL2 workspace uses a single shared variable, called DAT in this publication. (DAT is the name of the variable shared by SQL workspace facilities.)

The requirements for variables shared with AP 127 and AP 227 are:

Protocol	One variable The variable is used to pass requests to the auxiliary processor and to pass return codes and result data from auxiliary processor and SQL operations. The variable is assigned the operation code and any parameters required by that operation code. The structure of arguments to the operation codes is described in Arguments .
Maximum Number of Shared Variables	On mainframe systems, 10. On workstation systems, not limited by the auxiliary processor. See Multiple Shared Variables for more information.
Names	Any valid APL2 variable name of up to 255 characters.
Initial Values	Ignored.
Subsequent Values	Depend on the requested operations. See the detailed descriptions of the AP 127 and AP 227 operations in Auxiliary Processor Operations - Reference .
Access Control	0 1 1 0 is set by the auxiliary processor. 1 0 1 1 should be set by the user or workspace function. (This prevents a reference before the auxiliary processor has specified a result.) The SQL workspace function OFFER uses this protocol.

Shared Variable Access Sequences

After sharing has been established, an *operation code* can be passed to the auxiliary processor as a character vector. This character vector is the first item of an argument vector that is assigned to the shared variable DAT. A *reference* of DAT retrieves the result of the operation.

The recommended shared variable access procedures are:

- Synchronous (used by the SQL workspace)
 1. Specify a variable.
 2. Reference the variable and save the result.
 3. Process the data returned. Step 2 may be repeated for successive references.
 4. Repeat first three steps in sequence until processing is completed.
- Asynchronous
 1. Specify the variable with the first request.
 2. Reference the variable and save the result.
 3. Specify the variable with the next request.

4. Process the result of the reference at step 2.
5. Repeat steps 2 through 4 until processing is completed.

Multiple Shared Variables

Request Processing:

On mainframe systems, multiple shared variables are allowed, but transactions are processed for only one variable at a time. Once a request has been specified to a variable, all other variables are denied service, receiving a return code 115 until that variable has been retracted.

On workstation systems, each shared variable has its own connection to the database, and its own unit of work. Each variable's transactions are processed independently.

Scope of Options:

On mainframe systems, options and settings (from the SETOPT, ISOL, SSID, and TRACE requests) have an effect across all shared variables. For example, an event trace set by passing a TRACE request to any shared variable will trace events from all the shared variables.

On workstation systems, options and setting are independent for each shared variable. An event trace set by passing a TRACE request to DAT will trace only events for requests processed through DAT.

Arguments

The requests passed to AP 127 and AP 227 are in the following form:

code [*name*] [*stmt*] [*values*] [*options*]

where:

code

is an operation code, that specifies the auxiliary processor operation.

All AP 127 and AP 227 operations require an operation code as the first argument. This code is implied when the SQL workspace functions are used. [Auxiliary Processor Operations - Reference](#) contains a description of these codes.

name

is an SQL statement name that identifies the statement. Each name must be at least 2 characters long, and fit in 18 bytes of storage. Leading and trailing blanks are deleted.

Most AP 127 and AP 227 operations require a statement name as an argument.

stmt

is an SQL statement, which is represented as a vector or matrix of characters. The EXEC and PREP requests require an SQL statement as an argument.

The SQL statement presented as an argument to a PREP request can specify *vector-indexes* preceded by a colon. The indexes embedded in SQL statements refer to the position of values in a *value-list* passed in a subsequent CALL or OPEN request. These index references can appear in WHERE, SET, and VALUES clauses of SELECT, INSERT, UPDATE, and DELETE statements, and in the parameter lists of calls to stored procedures.

values

is a value-list that specifies values to be inserted in SQL statements that contain vector indexes. The value-list is passed as an argument of the CALL, PUT, and OPEN requests. It can be a vector or, for INSERT, UPDATE, and DELETE statements, a matrix. It must contain at least as many items (if a vector) or columns (if a matrix) as the value of the highest index specified in the SQL statement.

options

is a list of options which specifies the number of rows to be retrieved or the format of the result data. This information is passed as arguments to the FETCH or SETOPT requests.

Argument Data Structures

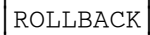
The shape of the arguments that may be specified to the shared variable is either simple or a vector of arrays.

The structure of the data passed to AP 127 and AP 227 is illustrated here using the defined function DISPLAY. DISPLAY, supplied by APL2 in the DISPLAY workspace, shows the shape and depth of arrays by surrounding them with boxes. (For a description of the DISPLAY workspace see *APL2 Programming: Using the Supplied Routines* on mainframe systems, or the *APL2 User's Guide* on workstation systems.)

Simple

The 'ROLLBACK' operation is one which can be issued with no arguments, so the array passed to the auxiliary processor is a simple character vector.

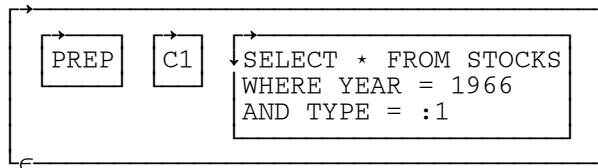
```
DISPLAY 'ROLLBACK'
```



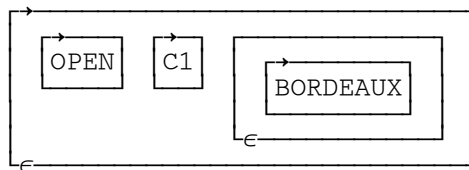
Vector of Arrays

The 'PREP' operation argument is a vector of arrays. The name 'C1' identifies the statement for subsequent operations. The SELECT statement includes a vector-index for a single-entry value-list passed with the 'OPEN' operation. The data is retrieved with a 'FETCH' operation, after which a 'CLOSE' operation can be issued for the statement 'C1'.

```
DISPLAY 'PREP' 'C1' QUERY1
```

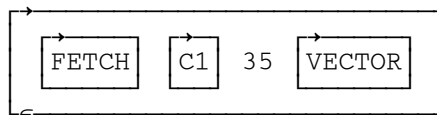


```
DISPLAY 'OPEN' 'C1' (<'BORDEAUX'>)
```

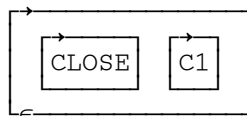


Note: 'BORDEAUX' is enclosed to present a value list with a single item.

```
DISPLAY 'FETCH' 'C1' 35 'VECTOR'
```



```
DISPLAY 'CLOSE' 'C1'
```



Results

The result returned from an auxiliary processor operation or SQL workspace function has the general format:

*r*code *data* [*stack*]

The result returned by AP 127 and AP 227 operations is always a two-item vector containing the following:

*r*code

is a five-item return code vector.

data

is the result data array. If no result data is required or if none is obtained because of an error, this item is an empty vector.

When the SQL workspace functions SQL, RESUME, and QUE are used, the result vector also contains a third item:

stack

is a request stack vector containing incomplete requests. This item is an empty vector if all requested operations have completed.

Return Code Vector

All APL2/SQL operations provide a five-item return code vector. The five items can be considered as a three-item completion code and a two-item error indicator as described in the following section.

Completion Code

The first three items contain a *completion code*. This code has five possible meanings, as follows:

0 0 0

Normal return

0 0 1

Normal return but table may not have been completely retrieved

0 1 0

Warning message

1 0 0

Error

1 1 0

Transaction backout

A completion code of 0 0 1 occurs on a FETCH request if the number of rows requested is less than or equal to the number of rows in the result table. The FETCH request retrieves table rows by moving the cursor one row at a time; it identifies the end of a result table after the cursor moves one row beyond the end. Thus, a FETCH request after this return code is received returns some rows or no rows, depending on the size of the table and the number of rows already retrieved.

The completion code 0 1 0 occurs, for example, if a FETCH request is issued when the current cursor position is beyond end-of-table or if a DELETE statement completes normally but deletes nothing. It is also issued when a value list is longer than the highest index value embedded in an SQL statement.

Note: When the completion code 1 1 0 (transaction backout) is returned, all changes made to the database since the last COMMIT or ROLLBACK request (or the beginning of the session if no COMMIT or ROLLBACK has been issued) have been discarded, and any locks acquired have been freed.

The term *transaction* is used here to refer to a *unit of recovery* or a *logical unit of work* - a sequence of statements that SQL treats as a single entity.

When transaction backout occurs, all cursors are freed.

Error Indicator

The error indicator consists of the last two items of the five-item return code.

The fourth item of the code vector indicates where the error or warning condition was detected:

- 0 No error or warning
- 1 Error or warning detected in the auxiliary processor
- 2 Error or warning detected in the database system
- 3 Error or warning detected in an SQL workspace function

If there is an error or warning, the fifth item gives an identifying message number.

More information about all three types of messages can be returned to the workspace through the defined function MESSAGE in the SQL workspace. The auxiliary processor 'MSG' operation returns messages only from the auxiliary processor and SQL (see [GET MESSAGE](#) for more information).

[APL2/SQL Interface Messages](#) lists the auxiliary processor and workspace function error messages.

The return code vector is summarized in [Return Code Vector](#).

Result Data Structures

The second item in the result is the data array. The contents of the result data array depend on which request was made.

Empty Data Item

The data array is empty if no rows are in the result of the SQL query or if it is an auxiliary processor command or SQL statement type that does not return a result, such as PREP or OPEN.

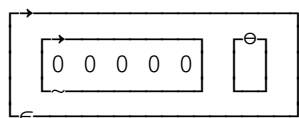
In the case of a FETCH request that returns no rows, the shape of the empty data array will be a null prototype of the result table. For MATRIX form, a matrix is returned. The shape is 0 by the number of columns in the table. For VECTOR form, a vector of null items, one per column, is returned. The type of each null item is the same as the type of the corresponding column.

In the case of requests such as PREP or OPEN without a value-list, which never return data, a simple character null is returned.

```

QUERY2
SELECT * FROM STOCKS WHERE
YEAR = 1971 OR YEAR = 1983
ORDER BY COST
RESULT←PREP 'C2' QUERY2
DISPLAY RESULT

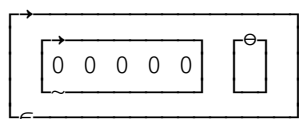
```



```

RESULT←OPEN 'C2'
DISPLAY RESULT

```



CALL, OPEN, PUT Result Data

For calls to stored procedures, the value-list passed on CALL is returned in its entirety in the data item. If any parameters were updated by the routine, the corresponding items of the value-list are updated. If the value-list item for the parameter is not large enough to contain the output data, it will be truncated.

For all other statements, if the entire value-list for a CALL, OPEN or PUT request is processed successfully, or no value-list is passed, the data item is a simple character null as described above.

If, however, an error occurs processing the value-list, the unprocessed data is returned in this item.

For example, if an ten-row matrix value-list is passed and an error occurs processing the fourth row, the fourth through the tenth rows are returned as a seven-row matrix. The application can choose to COMMIT or ROLLBACK the three rows that were processed successfully or correct the value-list and continue.

FETCH Result Data

The result of a successful FETCH request contains a nonempty array. The structure of the items of the result data vector can be one of two types, MATRIX or VECTOR.

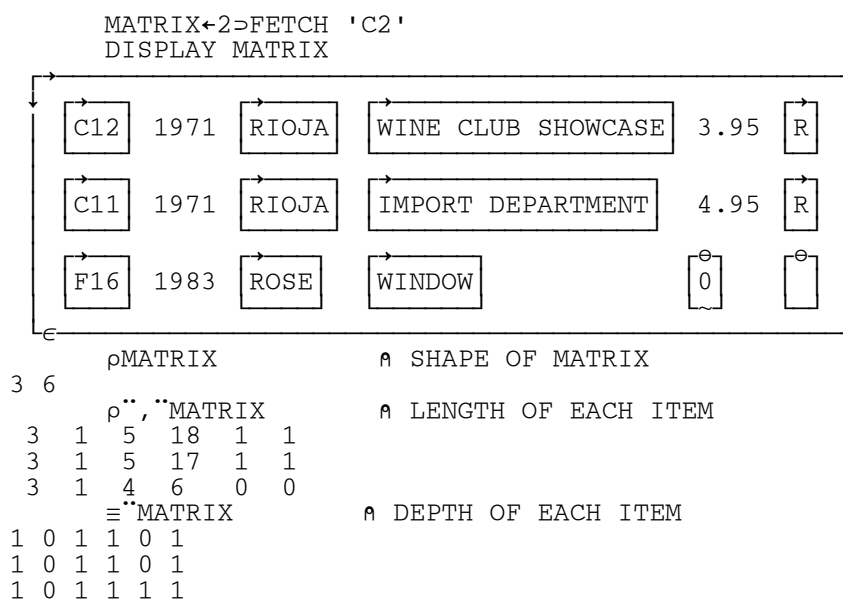
The structure type is controlled using the SETOPT request, or by passing an additional argument to the FETCH request.

Data Structure Type MATRIX

The default transfer data structure type is MATRIX, a nested matrix containing varying-length character fields, scalar numerics, and empty vectors.

- The shape of the matrix is r by c , where r is the number of rows returned by the FETCH request and c is the number of columns.
- Numeric fields are returned as numeric scalars.
SQL NULL values in numeric columns are returned as empty numeric vectors.
- Character fields are character vectors, including those of length 1. If the SQL data type is CHAR, these fields are all the same length.
SQL NULL values in character columns are returned as empty character vectors.

The following example uses the DISPLAY function to illustrate the structure of the result data.



Data Structure Type VECTOR

You may prefer to deal only with items of column data that are simple and homogeneous. You can accomplish this by specifying 'VECTOR' as an argument of the FETCH request or of a prior SETOPT request.

With the VECTOR option in effect, empty NULL values and length 0 character fields are replaced with zeros or blanks, character items are padded to the length of the longest item in the column vector, and all column vectors are disclosed as matrixes. Each result column is thus returned as a *homogeneous matrix*.

The shape of each simple matrix is r by c , where r is the number of rows returned and c is the column width. For numeric data, c is always 1. For character data, c is the width of the widest field in the column.

The depth of the vector is always 2; the depth of each item is always 1. The VECTOR result contains fewer items. It requires less space than the MATRIX result for the same table unless the table consists primarily of character fields that vary widely in length.

Data Representation Option LENGTH

If the data structure type is VECTOR, you can request a *length matrix* that gives the original length of each field. It is returned as an additional item in the result data vector.

You request the length matrix by specifying the option 'LENGTH' in a FETCH or prior SETOPT request.

The length matrix is a simple matrix with the same number of rows and columns as the result table. Numeric scalars are represented as having length 1. NULL fields have no length attribute in SQL; they are represented by a value of 0 in the length matrix.

The length option is included for use with the VECTOR format because the item as retrieved is no longer its original length, having been padded to the length of the longest item.

The next example illustrates the VECTOR result data and associated length matrix.

```
OPEN 'C2'
0 0 0 0 0
VECTOR←2>FETCH 'C2' 'VECTOR' 'LENGTH'
DISPLAY VECTOR
```

C12	1971	RIOJA	WINE CLUB SHOWCASE	3.95	R	3	1	5	18	1	1
C11	1971	RIOJA	IMPORT DEPARTMENT	4.95	R	3	1	5	17	1	1
F16	1983	ROSE	WINDOW	0		3	1	4	6	0	0

```
ρVECTOR          A SHAPE OF RESULT DATA VECTOR
7
≡VECTOR          A DEPTH OF RESULT DATA VECTOR
2
LENGTHMAT←(ρVECTOR) >VECTOR  A LENGTH MATRIX
LENGTHMAT
3 1 5 18 1 1
3 1 5 17 1 1
3 1 4 6 0 0
2>VECTOR        A SECOND ITEM
1971
1971
1983
```

For more illustrations of the data structures described in this section, see [Result Data from Workspace Functions](#).

Controlling the Number of Rows Returned

You can control the number of rows retrieved by specifying a number as an argument of the FETCH request, or by a prior SETOPT request. The default value is 20.

If you request fewer rows than the result table contains, the result table is returned as a series of table pieces, one piece being transferred with each FETCH request. Each piece consists of all columns and the next n rows of the result table following the last row returned.

If you request more rows than can fit in the available space, an error message is returned indicating that space is not sufficient for the number of rows requested. You can then reissue the FETCH request, specifying a smaller value for the number of rows.

Duration of the Options

When set by a FETCH request, these options override the current options established by a SETOPT request. They are effective only for the single FETCH request. When set by the SETOPT request, these options are effective for all subsequent FETCH requests unless explicitly overridden. When the shared variable is retracted, all options are reset to their default values.

Changing the Representation of Null Values

You may want to specify a character string to replace SQL NULL numeric values. In the following, the empty items in the result table is replaced by the character string 'NULL'.

```
MATNULL←MATRIX
((,MATNULL∈c'')/,MATNULL)←c'NULL'
((,MATNULL∈c'0')/,MATNULL)←c'NULL'
DISPLAY MATNULL
```

C12	1971	RIOJA	WINE CLUB SHOWCASE	3.95	R
C11	1971	RIOJA	IMPORT DEPARTMENT	4.95	R
F16	1983	ROSE	WINDOW	NULL	NULL

Operations on Result Data Arrays

Data from MATRIX Structures

The following examples show some expressions you can use to select data from a result data array of structure type 'MATRIX'. In the examples, the result data has been assigned to the variable MATRIX.

```
DISPLAY MATRIX
```

C12	1971	RIOJA	WINE CLUB SHOWCASE	3.95	R
C11	1971	RIOJA	IMPORT DEPARTMENT	4.95	R
F16	1983	ROSE	WINDOW	0	

Select Columns as Vectors

```
DISPLAY c[1]MATRIX[;2 5]
```

1971 1971 1983	3.95 4.95 0
----------------	-------------

Perform Calculations

```
COL5←MATRIX[;5]      A SELECT COLUMN 5
|/∈COL5              A MINIMUM OF COLUMN 5
3.95
+/∈COL5              A SUM
8.9
```

	ρCOL5	\mathbf{A} COUNT
3	$\text{CNTS} \leftarrow \rho\epsilon\text{COL5}$	\mathbf{A} COUNT EXCLUDING NULLS
2	CNTS	
	$+/\epsilon\text{COL5} \div \text{CNTS}$	\mathbf{A} AVERAGE
4.45		

The preceding expressions show the calculations that are the equivalent of SQL built-in functions. In addition, more complicated calculations can be performed in APL2.

Data from VECTOR Structures

The following example shows a way of selecting data from result tables of structure type 'VECTOR'. For the example, the length matrix has been dropped from the result.

```
VECTOR ← ⍒ 1 ↓ VECTOR
DISPLAY VECTOR
```

C12	1971	RIOJA	WINE CLUB SHOWCASE	3.95	R
C11	1971	RIOJA	IMPORT DEPARTMENT	4.95	R
F16	1983	ROSE	WINDOW	0	

Select Columns

```
DISPLAY ⍒, / VECTOR[2 5]
```

1971	3.95
1971	4.95
1983	0

Operation Dependencies

SQL statements can be:

- Executed immediately through an EXEC request.
- Prepared for later execution through a PREP request. The DECLARE and PREP requests name the statements so that they can be processed by subsequent operations.

The *state* of a named SQL statement consists of two numbers.

The first number indicates the type of the statement:

- | | |
|---|-------------------------|
| 0 | Type not yet determined |
| 1 | Non-SELECT |
| 2 | SELECT |
| 3 | Stored procedure call |

The second number indicates what stage of processing the statement has reached:

- | | |
|---|---|
| 0 | Statement not yet prepared. |
| 1 | Statement has been successfully prepared. |
| 2 | Statement has been successfully opened.
For stored procedures, a result set is ready for fetching. |

An SQL statement can be in any of ten different states:

State	Meaning
0 0	Statement is of undetermined type and has not yet been successfully prepared.
1 0	Statement was determined to be non-SELECT but was not successfully prepared.
1 1	Statement was determined to be non-SELECT and was successfully prepared.
1 2	Statement is non-SELECT and was successfully opened.
2 0	Statement was determined to be SELECT but was not successfully prepared.
2 1	Statement was determined to be SELECT and was successfully prepared.
2 2	Statement is a SELECT statement and was successfully opened.
3 0	Statement was determined to be a stored procedure call but was not successfully prepared.
3 1	Statement was determined to be a stored procedure call and was successfully prepared.
3 2	Statement is a stored procedure call and a result set is ready to be fetched.

AP 127 and AP 227 operations that require statements to be in a particular state are said to be *state-dependent* operations. Other operations are *state-independent*.

State-Dependent Operations

The following table summarizes the state-dependent operations. Each of these operations passes the name of an SQL statement as a character vector in DAT.

In the tables in this section, **I** and **O** represent **input** and **output** argument arrays:

- **I** indicates a workspace specification and an auxiliary processor reference.
- **O** indicates an auxiliary processor specification and a workspace reference.

Operation	Code	Shared Variable	Required State	Resulting State
CALL	'CALL'	I: name [values]	(1 1) Non-SELECT Prepared or (3 1) Procedure Call Prepared	Unchanged
CLOSE CURSOR	'CLOSE'	I: name	(n 2) Open	(n 1) Prepared
DECLARE	'DECLARE'	I: name ['HOLD']	Any	(0 0) Unprepared
DESCRIBE	'DESCRIBE'	I: name [type] O: table-description	(2 1) SELECT Prepared or (2 2) SELECT Open or (3 2) Procedure Call Open	Unchanged
FETCH	'FETCH'	I: name [options..] O: results	(2 2) Open	Unchanged
GET STATE	'STATE'	I: name O: state	Any	Unchanged
GET STMT	'STMT'	I: name O: stmt	Any	Unchanged
ODBC OPEN	'ODBCOPEN'	I: name type	Any	(2 2) SELECT Open
OPEN CURSOR	'OPEN'	I: name [values]	(n 1) Prepared	(n 2) Open
PREPARE	'PREP'	I: name stmt	Any	(n 1) Prepared
PURGE	'PURGE'	I: name or "	Any	None
PUT (CMS Only)	'PUT'	I: name values	(1 2) Non-SELECT Open	Unchanged

The relationships of state-dependent operations are summarized below:

This operation... Must precede...

```
'DECLARE'      'PREP', 'ODBCOPEN', 'STMT', 'STATE'
'PREP'         'DESCRIBE', 'CALL', 'OPEN', 'PURGE'
'OPEN'         'FETCH', 'PUT', 'CLOSE'
'ODBCOPEN'     'DESCRIBE', 'FETCH', 'CLOSE', 'PURGE'
```

Note: DECLARE is optional. The relationships stated must be obeyed only if it is used.

State-Independent Operations

The next table summarizes the state-independent operations. These operations are valid whenever a variable has been shared (overall state is shared).

Operation	Code	Shared Variable
<u>CONNECT</u>	'CONNECT'	I: [database identifier] O: [connection information]
<u>EXECUTE</u>	'EXEC'	I: stmt
<u>COMMIT WORK</u>	'COMMIT'	I: ['RELEASE']
<u>ROLLBACK WORK</u>	'ROLLBACK'	I: ['RELEASE']
<u>GET MESSAGE</u>	'MSG'	I: rcode O: message
<u>GET NAMES</u>	'NAMES'	O: names
<u>SET OPTIONS</u>	'SETOPT'	I: options
<u>GET OPTIONS</u>	'GETOPT'	O: options
<u>SET ISOLATION LEVEL</u>	'ISOL'	I: level
<u>GET ISOLATION LEVEL</u>	'ISOL'	O: level
<u>GET SQLCA</u>	'SQLCA'	O: sqlca contents
<u>GET SQLSTATE</u>	'SQLSTATE'	O: sqlstate value
<u>GET SSID</u> (TSO Only)	'SSID'	O: subsystem name
<u>SET SSID</u> (TSO Only)	'SSID'	I: subsystem name
<u>SET TRACE</u>	'TRACE'	I: (module level)
<u>GET TRACE</u>	'TRACE'	O: trace levels
<u>SET XML SIZE</u>	'XMLSIZE'	I: size
<u>GET XML SIZE</u>	'XMLSIZE'	O: size
<u>ODBC INFORMATION</u> (AP 227 Only)	'ODBC'	I: type O: info

Terminating the Unit of Work

When communicating with SQL, a *unit of work* is established with the first SQL call. A unit of work (sometimes known as a unit of recovery) defines to SQL a set of changes to the database that is recorded as one logical change.

A unit of work is terminated with a ROLLBACK or COMMIT. A ROLLBACK cancels the unit of work, and a COMMIT makes it permanent.

It is good programming practice to issue COMMIT and ROLLBACK as frequently as possible in order not to tie up resources that might be needed by other users. A responsible application takes steps to ensure that COMMIT and ROLLBACK are issued.

There are several methods that can be used to ensure responsible termination of the unit of work. One way is to localize the variable shared with the auxiliary processor in your functions. When the function terminates, the shared variable is retracted, and at that time a ROLLBACK is issued.

Another method for ensuring unit of work termination is to provide an *autocommit* facility for the end users. An autocommit facility is one in which the APL2 program issues COMMIT automatically after each logical set of SQL commands completes without errors. ROLLBACK is issued in the case of errors.

On workstations, some databases support a built-in autocommit facility. Where this is supported, it can be requested when the database connection is made. See [CONNECT](#) for more information.

Auxiliary Processor Operations - Reference

The following sections describe in detail each of the operations supported by AP 127 and AP 227.

For each operation, you specify the shared variable with an *input* vector, and reference it to retrieve an *output* vector.

The first item of the input vector is always the operation code. The remaining items, if they exist, are the arguments associated with the operation.

All output vectors consist of a five-item *return code vector* and a *data array*, as described in [Results](#).

The example that precedes each description gives the syntax for the specification of the shared variable, and one or more expressions for retrieving the output. Arguments are italicized and brackets indicate optional arguments. Double periods (..) indicate that more than one argument item may appear in the position.

The output description includes sample references of DAT with the data in DAT assigned to sample variables. DAT cannot be referenced more than once after being specified if \square SVC of 1 0 1 1 has been set; therefore, when more than one item is to be retained, vector specification is used to assign each item to an APL2 variable.

- [CALL](#)
- [CLOSE CURSOR](#)
- [COMMIT WORK](#)
- [CONNECT](#)
- [DECLARE](#)
- [DESCRIBE](#)
- [EXECUTE](#)
- [FETCH](#)
- [GET OPTIONS](#)
- [GET ISOLATION LEVEL](#)
- [SET ISOLATION LEVEL](#)
- [GET MESSAGE](#)
- [GET NAMES](#)
- [ODBC INFORMATION](#)
- [ODBC OPEN](#)
- [OPEN CURSOR](#)
- [PREPARE](#)
- [PURGE](#)
- [PUT](#)
- [ROLLBACK WORK](#)
- [SET OPTIONS](#)
- [GET SQLCA](#)
- [GET SQLSTATE](#)
- [SET SSID](#)
- [GET SSID](#)
- [GET STATE](#)
- [GET STMT](#)
- [SET TRACE](#)
- [GET TRACE](#)

- [SET XML SIZE](#)
- [GET XML SIZE](#)

CALL

Input: DAT \leftarrow 'CALL' *name* [*values*]

Output: (CODE *data*) \leftarrow DAT

Executes a data manipulation (INSERT, UPDATE, DELETE) SQL statement or stored procedure call processed using a previous PREP request.

name

Name of the SQL statement to be called.

values

Vector or matrix of substitution values. The vector, or each row of the matrix, is indexed using the index values provided during the PREP and passes the values to SQL for substitution into the statement. SQL is called once for a vector or once for each row of a matrix.

Each item in a value-list is *atomic*. That is, if numeric, it must be a single number or empty. If character, it must be a simple scalar, empty vector, or enclosed vector.

Note: Empty vectors are invalid for SQL clauses that disallow the NULL value.

data

For data manipulation statements, unprocessed data from the value-list. If all data was processed successfully or no value-list was passed, this item is a simple character null.

For stored procedure calls, the updated value-list. If any parameters were updated by the procedure, the corresponding items of the value-list are updated. If any value-list item is not large enough to contain its updated value, the data is truncated.

Required Initial State: Prepared (1 1) or (3 1)

Resulting State: Unchanged.

CLOSE CURSOR

Input: DAT ← 'CLOSE' *name*

Output: CODE ← ↑DAT

Closes an open cursor statement.

name

Name of the cursor statement to be closed.

Required Initial State: Open (3 2) or (2 2) or (1 2)

Resulting State: Prepared (3 1) or (2 1) or (1 1)

Usage Notes:

- If the statement referenced by *name* is a stored procedure call, this operation terminates access to all results sets build by the procedure.

COMMIT WORK

Input: DAT ← 'COMMIT' ['RELEASE']

Output: CODE ← ↑DAT

Makes permanent all changes made to the database since the last COMMIT or ROLLBACK request.

'RELEASE'

Specifies that the database connection should be released.

On CMS systems, a COMMIT WORK RELEASE is issued.

On TSO systems, a Call Attach Facility DISCONNECT is issued.

On workstation systems, SQLDisconnect is called.

All active statements are purged.

All locks are released.

Usage Notes:

- If the 'RELEASE' option is not specified, active statements are not purged. Cursors declared with the HOLD attribute will remain in position on some database systems. On these systems, cursors can be reused after COMMIT. On other systems, SQL errors result if the cursors are reused. See [DECLARE](#) for more information on the HOLD attribute.

CONNECT

On Workstation systems:

Input: DAT←'CONNECT' ['RESET' | {*database* | *driver*} [*id password*] ['AUTOCOMMIT']]

Output: (CODE INFO)←DAT
(*driver*)←INFO

On TSO DB2 systems:

Input: DAT←'CONNECT' [*id password*] ['RESET' | *database*]

Output: (CODE INFO)←DAT
(*server id database*)←INFO

On CMS SQL/DS systems:

Input: DAT←'CONNECT' [*id password*] [*database*]

Output: (CODE INFO)←DAT
(*server id database*)←INFO

Specifies the user ID or database server to be accessed, or, if issued without parameters, queries the status of the database connection.

'RESET'

On workstation systems, severs the current database connection.

On TSO systems, resets the connection to the default database.

database

Name of the database to be accessed

(CMS - 1 to 18 characters; TSO - 1 to 16 characters; Workstations - not limited by the auxiliary processor. The database and/or operating system may enforce stricter limitations.)

id

User id for database operations

(CMS - 1 to 8 characters; TSO - 1 to 255 characters; Workstations - not limited by the auxiliary processor. The database and/or operating system may enforce stricter limitations.)

password

Password associated with the user id

(CMS - 1 to 8 characters; TSO - 1 to 255 characters; Workstations - not limited by the auxiliary processor. The database and/or operating system may enforce stricter limitations.)

server

Server identifier. Begins with "ARI" on CMS, "DSN" on TSO.

driver

Database connection information string. The syntax of this string is data source-dependent. Some examples:

'DSN=APL2TEST'

'DBQ=Northwind.mdb;DRIVER={Microsoft Access Driver (*.mdb)}'

'AUTOCOMMIT'

Turns on database autocommit facility, if any.

Usage Notes:

APL2 Programming: Using SQL SH21-1057-07

© Copyright IBM Corporation 1984, 2008

- In previous implementations of AP 127 on workstations, a database connection mode of 'EXCLUSIVE' or 'SHARE' was accepted after the *database* parameter. For upward compatibility, these keywords will be tolerated, but ignored.
- On mainframe systems, if no CONNECT request is issued to AP 127, or the CONNECT request is issued without parameters, an implicit connection is established:
 - Under VM, the VM user ID and the database specified by the previous SQLINIT command are used.
 - Under TSO, the DB2 authorization ID is used.

On workstation systems, CONNECT is not optional. A CONNECT request is required to identify the Data Source to be accessed.

DECLARE

Input: DAT ← 'DECLARE' *name* ['HOLD' | 'NOHOLD']

Output: CODE ← ↑DAT

Defines a statement name and optionally assigns the HOLD attribute to it. SELECT statements declared with the HOLD attribute retain their cursor positions across COMMIT.

name

Name of the cursor statement to be declared.

'HOLD'

Specifies that the HOLD attribute should be applied to this statement name.

'NOHOLD'

Specifies that the HOLD attribute should not be applied to this statement name. (This is the default.)

Required Initial State: Any (n n)

Resulting State: Declared (0 0)

Usage Notes:

- There is a limit of 40 active names.
- DECLARE is optional. If it is not issued, a statement name will be automatically declared without the HOLD attribute at the time of the PREP or ODBCOPEN request. If issued, DECLARE must precede PREP or ODBCOPEN.
- Once a DECLARE is issued with HOLD, that attribute remains on the cursor name until another DECLARE is issued for the same name, or until the cursor is explicitly or implicitly purged (with a PURGE, COMMIT RELEASE, ROLLBACK, ROLLBACK RELEASE, or CONNECT RESET).
- Not all systems, and not all types of SQL statements, support the HOLD attribute. This operation has no effect if HOLD is not supported by the current database or statement. In that case, cursor behavior on COMMIT is data source-dependent.

DESCRIBE

Input: DAT \leftarrow 'DESCRIBE' *name* [*type*]

Output: (CODE *data*) \leftarrow DAT

Returns result table column information.

name

Name of a SELECT statement that has been processed using a PREP or ODBCOPEN request, or name of a stored procedure call statement that has been processed by a CALL request followed by an OPEN request.

type

One of the following:

'NAMES ' (return column names only)

'LABELS ' (return column labels only)

'BOTH ' (return both names and labels)

'ANY ' (return labels if they exist, otherwise return names)

If no type is specified, 'NAMES ' is the default.

data

Nested matrix containing column information.

The first row of the matrix contains the column names ('BOTH ' or 'NAMES ' option), column labels ('LABELS ' option) or a combination of names and labels ('ANY ' option). The second row gives the SQL data type attributes for the corresponding columns, and the third row gives the null specifications for the columns. If you specified the 'BOTH ' option, a fourth row is returned containing the column labels.

For example:

BIN	YEAR	TYPE	STORLOC	COST	COLOR
C 3	S	V 12	V 20	D 7 2	C 1
NOT NULL	NULL	NOT NULL	NULL	NULL	NULL

Required Initial State: SELECT prepared (2 1) or SELECT open (2 2) or Stored Procedure open (3 2)

Resulting State: Unchanged

Usage Notes:

- Not all systems support labels. If not supported, nulls are returned where the labels would be.
- The following table defines the abbreviations for data type used by the DESCRIBE request:

Code	SQL Data Type
O	BIT
T	TINYINT
S	SMALLINT

Code	SQL Data Type
I	INTEGER
B	BIGINT
D <i>m n</i>	Packed DECIMAL (<i>m,n</i>)
N <i>m n</i>	NUMERIC (<i>m,n</i>)
Z <i>m n</i>	Zoned DECIMAL (<i>m,n</i>)
R	REAL
F	FLOAT
C <i>n</i>	CHAR (<i>n</i>)
V <i>n</i>	VARCHAR (<i>n</i>)
L <i>n</i>	LONG VARCHAR
CL <i>n</i>	CLOB (<i>n</i>)
BI <i>n</i>	BINARY (<i>n</i>)
VB <i>n</i>	VARBINARY (<i>n</i>)
LB <i>n</i>	LONG VARBINARY
BL <i>n</i>	BLOB (<i>n</i>)
GR <i>n</i>	GRAPHIC (<i>n</i>)
VG <i>n</i>	VARGRAPHIC (<i>n</i>)
LG <i>n</i>	LONG VARGRAPHIC
DL <i>n</i>	DBCLOB (<i>n</i>)
WC <i>n</i>	WIDE CHAR (<i>n</i>)
WV <i>n</i>	WIDE VARCHAR (<i>n</i>)
WL <i>n</i>	WIDE LONG VARCHAR
TI	TIME
DT	DATE
TS	TIMESTAMP
RI	ROWID
X	XML

For more information, see [SQL and APL2 Data Types](#).

EXECUTE

Input: DAT \leftarrow 'EXEC' *stmt*

Output: CODE \leftarrow \uparrow DAT

Executes an SQL statement without parameters.

Mainframe systems: Equivalent to a Dynamic SQL EXECUTE IMMEDIATE.

Workstation systems: Equivalent to SQLExecDirect.

stmt

Any SQL statement except SELECT, CONNECT, COMMIT, and ROLLBACK.

Usage Notes:

- The SQL statement is executed exactly as it is written, with no argument substitution and no data transfer. Therefore, it cannot contain value-list index references.
- Stored procedure calls may be processed using this command. However, the following restrictions apply:
 1. All parameters to the procedure must be declared as type INPUT.
 2. No result sets built by the procedure will be available.

Stored procedure calls with INOUT or OUTPUT parameters, or with result sets, should be processed using the PREPARE and CALL requests.

FETCH

Input: DAT \leftarrow 'FETCH' *name* [*options..*]

Output: (CODE *data*) \leftarrow DAT

Returns new result table data as the second item of the shared variable result.

name

Name of an open cursor statement.

options

Any one or all of the following items in any order:

- A number that specifies the maximum number of rows to be retrieved.
- A character vector indicating the data transfer option 'MATRIX' or 'VECTOR'.
- A character vector indicating the length matrix option 'LENGTH' or 'NOLENGTH'.

'LENGTH' is ignored if 'VECTOR' is not also specified or already in effect from a previous 'SETOPT' operation.

The values of these options are valid only for the duration of the FETCH request. The options may be permanently set by a SETOPT request.

data

SQL table rows. The data is returned as either a matrix or a vector of simple matrices, depending on the data option chosen.

If the structure type is MATRIX (the default), the length of items of the result data matrix may vary.

If the structure type is VECTOR, each item of the result data vector is an r by c matrix, where r is the number of rows and c is the column width. For numeric data, the column width is always 1.

Required Initial State: SELECT Open (2 2) or Stored Procedure Open (3 2)

Resulting State: Unchanged

Usage Notes:

- At the end of the operation, if the cursor has not been moved past the last row of the table, a FETCH INCOMPLETE return code (0 0 1 0 0) is generated as a signal that there may be more data to be retrieved.

GET OPTIONS

Input: DAT ← 'GETOPT'

Output: (CODE *options*) ← DAT

Returns the current settings of the auxiliary processor options.

options

Three-item vector containing the current settings of the options-list:

1. 'MATRIX' or 'VECTOR' (result data format)
2. 'LENGTH' or 'NOLENGTH' (length matrix indicator)
3. Number of rows to be returned from the result table.

Usage Notes:

- The options-list is initialized to ('MATRIX' 'NOLENGTH' 20).
- For more information about the effects of these options, see [SET OPTIONS](#)

GET ISOLATION LEVEL

Input: DAT ← 'ISOL'

Output: (CODE *setting*) ← DAT

Queries the isolation level.

setting

Character vector containing the current isolation level ('CS', 'RR', 'RS' or 'UR'.)

Usage Notes:

- Not all ODBC systems support setting an isolation level. On systems where it is not supported, the ISOL setting has no effect.
- See [Concurrent Update of Shared Tables](#) for a discussion of ISOLATION LEVEL.

SET ISOLATION LEVEL

Input: DAT \leftarrow 'ISOL' *setting*

Output: CODE \leftarrow \uparrow DAT

Sets the isolation level.

setting

The setting must be one of:

- 'CS' for *Cursor Stability* (ODBC: *Read Committed*)
- 'RR' for *Repeatable Read* (ODBC: *Serializable*)
- 'RS' for *Read Stability* (ODBC: *Repeatable Read*)
- 'UR' for *Uncommitted Read* (ODBC: *Read Uncommitted*)

Usage Notes:

- The change in isolation level may not happen immediately. The requested isolation level is not set until the next time the database connection is requested. The connection is released on COMMIT RELEASE, ROLLBACK RELEASE and CONNECT RESET.
- On mainframe systems, it is possible to set the default isolation level for the entire session through the APNAMES parameter during APL2 invocation:

```
APNAMES (AP2X127('ISOL(setting)'))
```

On workstation systems, it is possible to set the default isolation level for the entire session through the -isol invocation parameter:

```
apl2 -isol setting
```

or by adding an ISOL keyword definition to the [Invocation Options] section of the apl2.ini file:

```
[Invocation Options]  
ISOL=setting
```

or by setting environment variable APLISOL before starting APL2:

```
SET APLISOL=setting
```

- If no isolation level is explicitly provided by the application or an invocation parameter, a default level of 'RR' is used.
- Not all ODBC systems support setting an isolation level. On systems where it is not supported, the ISOL request is accepted, but ignored.
- See [Concurrent Update of Shared Tables](#) for a discussion of ISOLATION LEVEL.

GET MESSAGE

Input: DAT \leftarrow 'MSG' *rcode*

Output: (CODE *msg*) \leftarrow DAT or
(CODE *msg text*) \leftarrow DAT

Returns the error message information associated with a return code.

rcode

Return code vector from a previous operation.

If the fourth item of *rcode* is 1, the fifth item can be any auxiliary processor error code.

If the fourth item of *rcode* is 2, the information and return codes from the most recent SQL operation are returned. (An error need not have occurred.)

msg

If the message is from the auxiliary processor, a character vector containing message text. The last associated message tokens are filled in. If the message has not been issued, associated tokens are filled with asterisks (**).

On mainframe systems, a 6 by 1 matrix containing the following six items:

1. SQLCODE (integer)
2. SQLERRM message tokens from the SQLCA
3. SQLERRP routine name
4. SQLERRD return codes (a six-item vector)
5. SQLWARN, an 11-character vector of warning indicators
6. SQLSTATE, a 5-character vector

For a definition of these fields, see the *SQL Reference* for your DB2 system.

On workstation systems, a five-element character vector containing the SQLSTATE.

text

Returned only if the fourth item of *rcode* is 2. On CMS systems, this item contains symptom strings for service as formatted by program ARISSMA. On all other systems, this item contains the text of the message associated with the given SQLSTATE, as formatted by the database system.

GET NAMES

Input: DAT \leftarrow 'NAMES'

Output: (CODE *names*) \leftarrow DAT

Returns a list of active statement names.

names

Vector of character vectors containing the names of all active statements.

Usage Notes:

- Using the results from this operation, the STATE request can be used to determine the state of a statement, and the STMT request can be used to retrieve a statement.
- If no names have been defined, a null name list is returned, and a warning code is issued.

ODBC INFORMATION

Input: DAT ← 'ODBC' *type*

Output: (CODE *info*) ← DAT

Retrieves information about the ODBC system.

Note: This command is valid for AP 227 only.

type

One of the following keywords:

'DATASOURCES' to request the output from the ODBC service *SQLDataSources*. This service returns a list of the available ODBC Data Sources (databases) and their associated drivers.

'DRIVERS' to request the output from the ODBC service *SQLDrivers*. This service returns a list of the installed ODBC Drivers and their attributes.

info

The requested information.

For more information on the ODBC services called by this command, consult the ODBC documentation for your database or ODBC driver manager. On-line information is available by searching for ODBC at <http://www.microsoft.com>.

Required Initial State: Any

Resulting State: Unchanged

ODBC OPEN

Input: DAT \leftarrow 'ODBCOPEN' *name type*

Output: CODE \leftarrow \uparrow DAT

Sets up an SQL table containing information about the currently connected ODBC database. Rows of the table can be retrieved by subsequent FETCH requests. The DESCRIBE, CLOSE, STMT, STATE and PURGE operations can also be used on the resulting cursor. It is equivalent to a cursor established by the PREP and OPEN operations.

Note: This command is valid for AP 227 only.

name

Identifies the name to be used for the cursor.

The argument *name* links this statement with subsequent FETCH, DESCRIBE, CLOSE, STMT, STATE and PURGE requests, and with previous DECLARE requests.

type

One of the following keywords:

'GETTYPEINFO' to request the output from the ODBC service *SQLGetTypeInfo*. This service builds a table with one row for each SQL data type supported by the currently connected data source. Each row contains the data type name, code, and other information about how it is defined. The number and content of the columns may vary between data sources.

'TABLES' to request the output from the ODBC service *SQLTables*. This service builds a table with one row for each SQL table, view, and schema defined in the currently connected data source. Each row contains the table name, type, and other information about how it is defined. The number and content of the columns may vary between data sources.

For more information on the ODBC services called by this command, consult the ODBC documentation for your database or ODBC driver manager. On-line information is available by searching for ODBC at <http://www.microsoft.com>.

Required Initial State: Any

Resulting State: SELECT Open (2 2)

OPEN CURSOR

Input: DAT \leftarrow 'OPEN' *name* [*values*]

Output: CODE \leftarrow \uparrow DAT

Opens a previously prepared SQL statement.

name

Name of a previously prepared SELECT or, on CMS, INSERT, statement, or the name of stored procedure call processed by PREP and CALL.

values

Vector of values to be selected by vector-indexes passed as part of the prepared statement, if a SELECT. (For an INSERT statement, the value-list is passed at the PUT step. For a stored procedure call, the value-list is passed at the CALL step.)

An item in *values* may be scalar, an empty vector, or an enclosed vector. Simple items, unless simple scalars, must be enclosed.

Required Initial State: Prepared (n 1)

Resulting State: Open (n 2)

Usage Notes:

- Upon successful completion of an OPEN request, the SELECT result table is established or the stored procedure result set is accessed. The current position of the cursor is just before the first row in the result table or result set. Rows can be retrieved by subsequent FETCH requests.

PREPARE

Input: DAT \leftarrow 'PREP' *name stmt*

Output: CODE \leftarrow \uparrow DAT

Prepares an SQL statement for later execution by a CALL or OPEN request.

name

Identifies the SQL statement to be prepared.

The argument *name* links this statement with subsequent CALL, PUT, OPEN, FETCH, CLOSE, DESCRIBE, and PURGE requests, and with previous DECLARE requests.

stmt

Any SELECT, INSERT, UPDATE, DELETE or stored procedure call can be prepared.

The statement may or may not contain APL2 vector-indexes. For example:

```
INSERT INTO STOCKS
(BIN, YEAR, TYPE, STORLOC, COST, COLOR)
VALUES ('I11', 1979, :1, 'ANNEX', :2, 'R')
```

Required Initial State: Any

Resulting State:

Prepared (3 1) or (2 1) or (1 1) if the PREPARE operation completes successfully; (3 0) or (2 0) or (1 0) if it does not complete successfully.

Usage Notes:

- There is a limit of 40 active names.

PURGE

Input: DAT \leftarrow 'PURGE' *name*

Output: CODE \leftarrow \uparrow DAT

Removes one statement or all statements from the list of active names.

name

Name of statement to be removed, or ' ' (empty character vector) to remove all active statement names.

All traces of the statement are removed, and storage with which it is associated is freed. If the statement is open, it is closed before being purged.

No COMMIT or ROLLBACK is issued.

Required Initial State: Any

Resulting State: Unprepared

PUT

Input: DAT \leftarrow 'PUT' *name values*

Output: (CODE *data*) \leftarrow DAT

Valid on CMS systems only.

PUT executes an INSERT statement processed by previous PREP and OPEN requests. This differs from CALL in that the database system can internally enhance performance by using blocking, and in that the value-list is required.

name

Name of the SQL statement to be called.

values

Vector or matrix of substitution values. The vector, or each row of the matrix, is indexed using the index values provided during the PREP, and the value is passed to SQL for substitution into the statement. SQL is called once for a vector or each row of a matrix.

Each item in a value-list is atomic. That is, if numeric, it must be a single number or empty. If character, it must be a simple scalar, empty vector, or enclosed vector.

data

Unprocessed data from the value-list. If all data was processed successfully, this item is a simple character null.

Required Initial State: Open (1 2)

Resulting State: Unchanged

ROLLBACK WORK

Input: DAT ← 'ROLLBACK' ['RELEASE']

Output: CODE ← ↑DAT

Backs out all changes made to the database since the last COMMIT or ROLLBACK request. The state of all prepared statements is reset to unprepared. All locks are released.

'RELEASE'

Specifies that the database connection should be released.

On CMS systms, a COMMIT WORK RELEASE is issued.

On TSO systems, a Call Attach Facility DISCONNECT is issued.

On workstation systems, SQLDisconnect is called.

SET OPTIONS

Input: DAT \leftarrow 'SETOPT' *options..*

Output: CODE \leftarrow \uparrow DAT

Sets the values of the auxiliary processor options.

The options established replace the options-list currently in effect. They remain in effect until they are set again, or until the shared variable is retracted.

They can be dynamically overridden by the options-list argument of a FETCH request.

options

Any one or all of the following items in any order:

- A number that specifies the maximum number of rows to be retrieved.
- A character vector indicating the data transfer option 'MATRIX' or 'VECTOR'.
- A character vector indicating the length matrix option 'LENGTH' or 'NOLENGTH'.

Usage Notes:

- If 'LENGTH' is specified and 'VECTOR' has not been specified (either in this options-list or in the FETCH request options-list), no length matrix will be returned.
- If the list contains conflicting options, the last named is chosen.
- The options-list is initialized with the values ('MATRIX' 'NOLENGTH' 20).

GET SQLCA

Input: DAT \leftarrow 'SQLCA'

Output: (CODE *sqlca*) \leftarrow DAT

Retrieves the current contents of the SQL communications area (SQLCA).

After an SQL operation, this control block contains return codes and, if an error has occurred, additional diagnosis information.

sqlca

A 6 by 1 matrix containing the following six items:

1. SQLCODE (integer)
2. SQLERRM message tokens
3. SQLERRP routine name
4. SQLERRD return codes (a six-item integer vector)
5. SQLWARN, an 11-character vector of warning indicators
6. SQLSTATE, a 5-character vector

For a definition of these fields, see the *SQL Reference* for your DB2 system.

Usage Notes:

- The SQLCA is not actually used internally on workstation systems. For compatibility, however, the workstation auxiliary processors store their diagnostic information in a similar structure, and the same array structure is returned by this operation code.
 - The SQLCODE item contains the database return code.
 - The SQLERRM item contains the database error message text, if any.
 - The SQLSTATE item contains SQLSTATE.
 - The SQLERRP and SQLWARN items are not used, and will always contain blanks.
 - The SQLERRD vector is also not used, but for compatibility with mainframe AP 127, the number of rows affected by data manipulation statements (INSERT, UPDATE, DELETE) is obtained and stored in the third item of SQLERRD after each EXEC and CALL operation. The other items of SQLERRD will always contain 0.

GET SQL STATE

Input: DAT \leftarrow 'SQLSTATE'

Output: (CODE *sqlstate*) \leftarrow DAT

Retrieves the current contents of the SQLSTATE variable. SQLSTATE is a status indicator that is common on all SQL relational database systems.

sqlstate

A 5-element character vector containing the current contents of SQLSTATE.

SET SSID

Input: DAT ← 'SSID' *setting*

Output: CODE ← ↑DAT

Valid in TSO only.

Sets the DB2 subsystem ID.

setting

Character vector containing the subsystem ID to be accessed. Limited to four characters in length.

Usage Notes:

- This command can only be issued when the database connection is not active, and takes effect the next time the connection is made. (The database connection is inactive when APL2 is first invoked. Once the connection is made, it can be made inactive by retracting the shared variable, or by issuing the ROLLBACK RELEASE, COMMIT RELEASE or CONNECT RESET requests.)
- It is possible to set the default subsystem ID for the entire session through the APNAMES parameter during APL2 invocation:
APNAMES (AP2X127('SSID(*system*) '))
- If no subsystem id is ever specified with this command or APNAMES, the default value of 'DSN' is used.

GET SSID

Input: DAT \leftarrow 'SSID'

Output: (CODE *setting*) \leftarrow DAT

Valid in TSO only.

Queries the DB2 subsystem ID

setting

Character vector containing the current subsystem ID

GET STATE

Input: DAT \leftarrow 'STATE' *name*

Output: (CODE *state*) \leftarrow DAT

Yields the current state for an SQL statement.

name

Name of the SQL statement for which the state is desired.

state

Two-element numeric vector with the statement state.

See [Operation Dependencies](#) for information on the possible values for state.

Required Initial State: Any state (n n).

Resulting State: Unchanged

Usage Notes:

- A list of active statement names can be obtained with the NAMES request.
- An error indication is returned if the statement *name* does not exist (has not been declared or prepared).

GET STMT

Input: DAT \leftarrow 'STMT' *name*

Output: (CODE *stmt*) \leftarrow DAT

Yields a specified SQL statement.

name

Name of the SQL statement to be returned.

stmt

Value of the SQL statement as originally passed on the PREP request.

Required Initial State: Any state (n n).

Resulting State: Unchanged

Usage Notes:

- A list of active statement names can be obtained with the NAMES request.
- An error indication is returned if the statement *name* does not exist (has not been declared or prepared).

SET TRACE

Input: DAT \leftarrow 'TRACE' (*modnum level*) . . .

Output: CODE \leftarrow \uparrow DAT

Sets tracing levels.

modnum

Module number for tracing. Module number 0 specifies all modules. Other module numbers vary by system.

level

Trace level number (0 to 9). Trace level 0 specifies no tracing. Trace level 9 specifies the maximum level of tracing.

Usage Notes:

- If multiple conflicting settings are passed the last (rightmost) setting passed will be used.
- Your IBM Support personnel will provide the appropriate module and level numbers for your system when necessary for debugging purposes.

GET TRACE

Input: DAT ← 'TRACE'

Output: (CODE *levels*) ← DAT

Returns the current trace settings.

levels

Numeric vector with trace settings (0 to 9) for each auxiliary processor module. The length of this vector varies by system.

SET XML SIZE

Input: DAT \leftarrow 'XMLSIZE' *size*

Output: CODE \leftarrow \uparrow DAT

Sets the size used for fetching XML data items.

size

The number of bytes of storage to be allocated for fetching each XML data item. XML data items longer than the specified size will be truncated.

If no size is specified by the application, the default size is 65536 bytes.

GET XML SIZE

Input: DAT \leftarrow 'XMLSIZE'

Output: (CODE *size*) \leftarrow DAT

Returns the current size for fetching XML data items.

size

The number of bytes of storage allocated for each XML data item.

The SQL Workspace

You can use the facilities of the SQL workspace to make passing requests to AP 127 and AP 227 easier. These requests can be for the execution of individual SQL statements, or they can be to set options for and retrieve information from the auxiliary processor.

You can copy the SQL workspace functions into your application workspace and use them as written, or you can use them as models to create your own customized workspaces for SQL access.

The SQL workspace functions can be used with either AP 127 or AP 227. The variable `SQL_AP` controls which auxiliary processor will be accessed. Its default value is 127. To use the functions with AP 227, set `SQL_AP` to 227.

The SQL workspace contains the following categories of defined functions and operators:

- [Data Access Functions](#)
- [Task Control Functions](#)
- [User Support Functions](#)
- [Defined Operator UNTIL](#)
- [Input and Evaluation Functions](#)
- [Iterative Processing Functions](#)
- [Charting Functions](#)

The next sections contain a summary of these functions, and some examples of their usage. Detailed descriptions of their arguments and results are contained in [The SQL Workspace - Reference](#).

Data Access Functions

The data access functions access SQL tables.

These functions are the same as their corresponding auxiliary processor operations in the arguments they require and in the results they produce, except that operation codes are implied by the function calls and all specifications and references of the shared variable are handled by the functions.

The following table lists each data access function and its purpose.

Name	Purpose
<u>EXEC</u>	Executes an SQL statement with no arguments.
<u>DECLARE</u>	Assigns attributes to a cursor name.
<u>PREP</u>	Prepares a data manipulation statement, a SELECT statement or a stored procedure call.
<u>CALL</u>	Invokes a data manipulation statement or stored procedure call with arguments.
<u>OPEN</u>	Initializes a cursor on a prepared SELECT statement or stored procedure result set.
<u>ODBCOPEN</u>	Builds a result table containing ODBC database information and initializes a cursor for the table. (AP 227 only)
<u>FETCH</u>	Retrieves a specified number of rows of a result table and optionally sets the structure of the result table.
<u>PUT</u>	Invokes an INSERT statement with blocking (CMS only)
<u>CLOSE</u>	Closes a cursor statement. The cursor position is lost.
<u>DESC</u>	Returns result table column information.

Task Control Functions

The task control functions manage the auxiliary processor environment.

The OFFER function offers to share a variable named DAT with the auxiliary processor indicated by the variable SQL_AP. This function may be called explicitly, or it will be implicitly called by the other functions when needed.

The remaining task control functions are the same as their corresponding auxiliary processor operations in the arguments they require and in the results they produce, except that operation codes are implied by the function calls and all specifications and references of the shared variable are handled by the functions.

Name	Purpose
OFFER	Offers to share a variable.
CONNECT	Connects explicitly to the database.
COMMIT	Commits a logical unit of work.
ROLLBACK	Cancels a logical unit of work.
SETOPT	Sets options-list values.
GETOPT	Returns options-list values.
NAMES	Returns a list of active statement names.
SQLCA	Returns current contents of SQLCA (SQL communications area).
SQLSTATE	Returns current value of SQLSTATE error indicator.
STATE	Returns state of named SQL statement.
STMT	Returns named SQL statement.
PURGE	Removes a statement from the list of prepared statements.
ISOL	Queries or sets the isolation level.
SSID	Queries or sets the subsystem ID. (TSO Only)
TRACE	Enables or disables auxiliary processor tracing.
XMLSIZE	Sets or queries the size used for fetching XML data items.

User Support Functions

The user support function ease the task of using the auxiliary processor interface by allowing sets of commands to be executed together, and by retrieving error information.

Name	Purpose
QUE	Processes a request stack and returns all result data.
RESUME	Processes a request stack and returns only FETCH data.
SQL	Processes SQL statements by creating a request stack.
SHOW	Displays a result vector in tabular format.
SQLHELP	Fetches help text information from database tables (CMS only).
MESSAGE	Provides error message information.
ODBC	Retrieves ODBC system information (AP 227 only).

- The defined function QUE accepts and processes in a single function call a vector of operations (a *request stack*) such as 'PREP', 'OPEN', 'FETCH', and 'CLOSE'.
- The RESUME function takes a stack of data access operations, calls QUE to process them, and reduces the second item of the result to an array that contains only the result of FETCH requests.
- The SQL function accepts an SQL statement as an argument, determines the appropriate sequence of operations to process the statement, and calls the RESUME function to execute the request stack.

For query (SELECT) statements, SQL forms a request stack in the sequence: 'PREP' 'OPEN' 'FETCH' 'CLOSE'. For data manipulation (INSERT, UPDATE, DELETE) and stored procedure call statements, SQL forms a request stack in the sequence: 'PREP' 'CALL'. The AP 227 ODBCOPEN operation types (GETTYPEINFO and TABLES) are recognized as special statements, and a request stack of 'ODBCOPEN' 'FETCH' 'CLOSE' is generated. COMMIT and ROLLBACK statements are handled as separate requests. For all other statements, SQL forms an 'EXEC' request.

Note: Because the SQL function must be common to all operating systems, it does not use the 'PREP' 'OPEN' 'PUT' sequence for INSERT statements. If you are running on CMS only, you may want to write your own function that uses 'PUT', or use it directly, to gain the performance advantage it provides.

If a matrix argument for a value-list is passed to the SQL function for a SELECT statement it creates the sequence of iterated requests required to process all rows of the matrix. Thus for a SELECT statement with two rows of search conditions the SQL function forms the sequence: 'PREP' 'OPEN' 'FETCH' 'CLOSE' 'OPEN' 'FETCH' 'CLOSE'. Successful completion of the second sequence returns a *result data array* that is a 2 by 1 nested matrix containing the data from each 'FETCH' operation.

If the last return code contains a value not listed in the global variable CODE_, QUE, RESUME, and SQL suspend request processing. They return the return code vector, any result data obtained up to the time of the error, and the remainder of the request stack. The error can then be corrected and the RESUME function called to process the remainder of the stack.

Note: The results of stacked 'FETCH' operations are *accumulated*. They do not emulate a subquery.

- The SHOW function reshapes the result vector so that its three items display down the screen rather than across.

SHOW is also used to display error messages. The SHOW function calls the function MESSAGE, which replaces the return code with a completion code description such as 'TRANSACTION BACKOUT', followed by the error message. The MESSAGE function replaces an all-zero return code vector with an empty vector. Thus, on a normal return from a request stack that includes 'FETCH' operations, only result data tables are visible in the SHOW function display; the other two items (return code and remaining stack) are empty.

- The SQLHELP function shows how to retrieve help text available in tables on a CMS system. This function can be used directly or modified to fit into your application. The variable HELPQUERY contains the SQL statement needed to retrieve the help text.
- The MESSAGE function can be executed directly to return messages when the return code vector contains an error code from a workspace function or a nonzero return code from the auxiliary processor.
- The ODBC function retrieves ODBC system information, such as names of available ODBC data sources and drivers.

Defined Operator UNTIL

The defined operator UNTIL is used to create the derived function (F UNTIL) STACK, which processes each item of the stack using the function F. UNTIL terminates stack processing when a nonzero return code is returned from the calling function.

UNTIL is normally called by the workspace function QUE, but it can also be used to process a vector of SQL function arguments. The following expression accumulates the result vectors from the SQL function *until* an error occurs or the list is exhausted.

```
SQL UNTIL statements-list
```

statements-list is a vector of SQL function arguments, as illustrated in the following example:

```
SQL UNTIL CWINE (IWINE WVALS) 'COMMIT'
```

Each variable (CWINE, IWINE, WVALS, and so on) has been assigned an SQL statement or a value-list to be passed as an argument to an auxiliary processor operation. For the statements associated with each variable, see [A Beginning Example](#)

Input and Evaluation Functions

The evaluation functions make creating data to be passed to AP 127 and AP 227 easier.

Name	Purpose
DATE	Accepts a numeric time vector in APL2 format and returns a character string formatted for the SQL data type DATE.
TIME	Accepts a numeric time vector in APL2 format and returns a character string formatted for the SQL data type TIME.
TIMESTAMP	Accepts a numeric time vector in APL2 format and returns a character string formatted for the SQL data type TIMESTAMP.
IN	Accepts character input and returns a matrix.
EVAL	Evaluates a matrix that contains mixed character and numeric data.
EVALSIM	Evaluates a mixed matrix using descriptors of lengths and types.

Iterative Processing Functions

The iterative processing functions are used to fetch and process rows of large SQL tables in steps of manageable size.

Name	Purpose
QUERY	Calls the SQL function until all data of a table is fetched, and appends the column names to the top of the table.
PROCEDURE	Calls the SQL function to process a stored procedure call, then fetches all available result sets in the same manner as QUERY.
REPORT	User-modifiable reporting function.
HEAD	Appends a description to a table.
APPEND	Appends a table to a table.
COMBINE	Combines fetched data.

Charting Functions

The charting functions extract data from relational tables and pass it to the [GDDM](#) Interactive Chart Utility. (Mainframe systems only).

Name	Purpose
CHARTDATA	Prepares input for the CHART function.
CHART	Retrieves SQL data and displays it with the ICU.

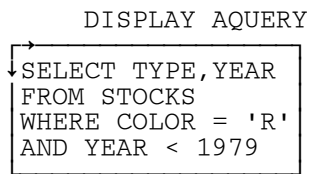
User Support Function Examples

The arguments passed to the SQL and QUE functions may be:

- Only the SQL statement to be executed
- SQL statement and a simple vector of values (value-list)
- SQL statement and a matrix value-list

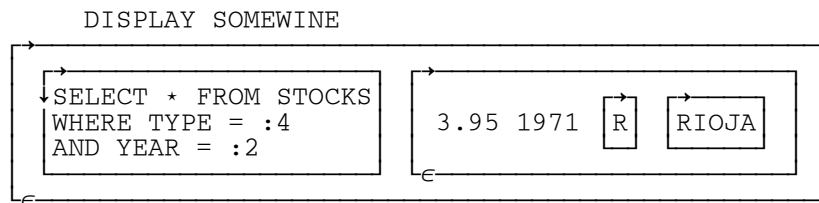
SQL Statement Only

The variable AQUERY is an SQL statement with values embedded in the statement.

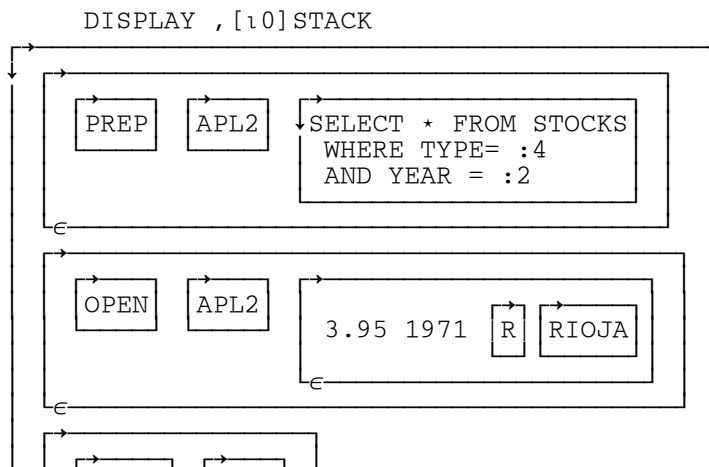


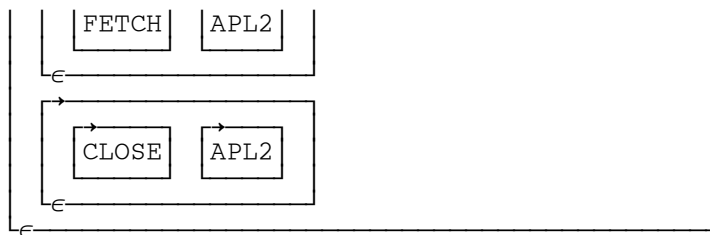
Vector Value-List

The argument to the SQL function can contain the SQL statement with vector-indexes and a value-list in the form of a vector. The value-list can contain more items than are actually indexed, and the values can be indexed in any order, as illustrated below.



Based on the argument it receives, the SQL function creates a stack of auxiliary processor operations. For the argument SOMEWINE, the request stack is a 4-item vector of items with varying length.





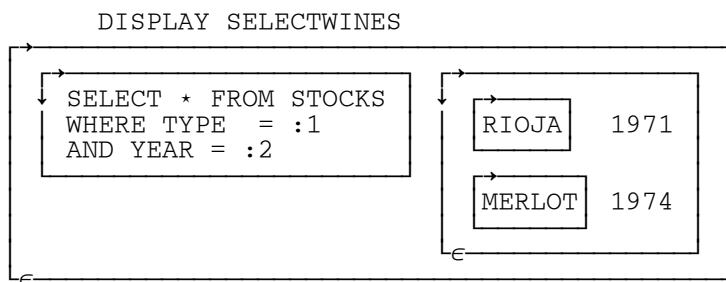
The selection is based only on the fourth and second items in the list. The auxiliary processor ignores the rest.

Matrix Value-List

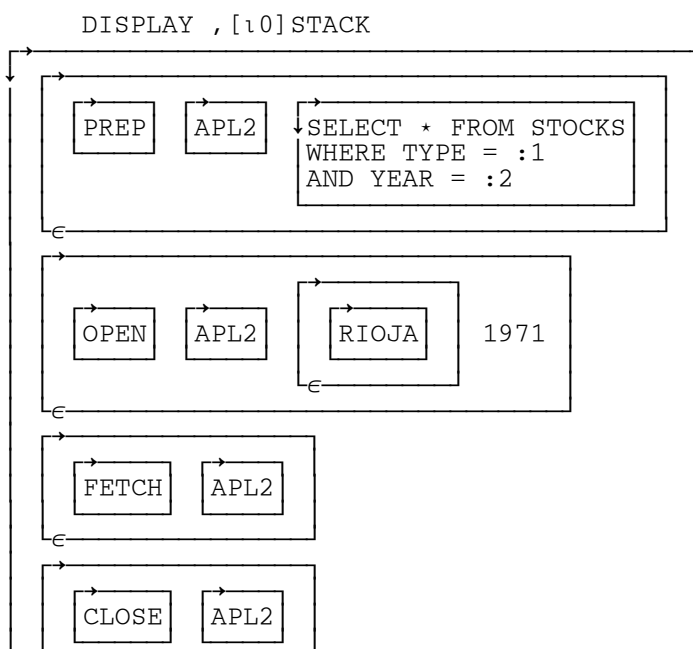
When the SQL function receives a matrix value-list for a SELECT statement it indexes each row of the matrix as a separate vector.

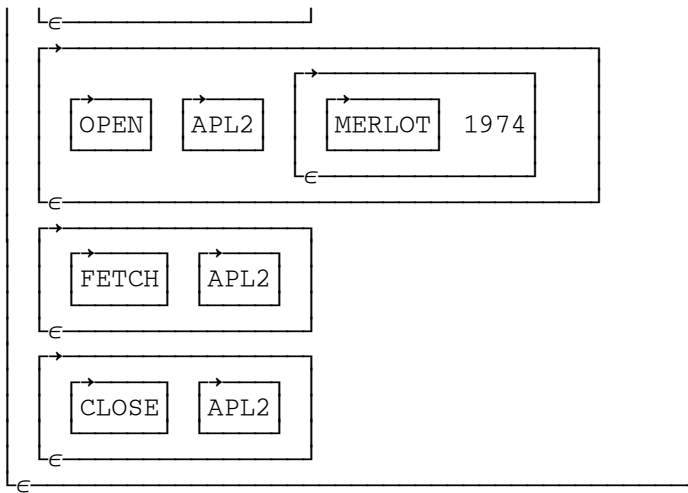
The variable SELECTWINES contains a SELECT statement and a matrix value-list.

SELECTWINES is a vector of length 2. The first item is an SQL statement. The second is a value-list that is a 2 by 2 matrix:



The stack of auxiliary processor operations that the SQL function creates for this SELECT statement with this matrix value-list is:





The result data arrays of all the SELECT statements illustrated in the preceding section are illustrated in the following section.

Result Data from Workspace Functions

The result of a call to the SQL function is always a vector of length 3. The third item is empty if the request is completed.

The call to the SQL function may use a variable, like AQUERY, which contains the SQL statement to be processed.

```
TAB←SQL AQUERY
```

The statement can also be entered dynamically. In the following illustration, the SQL workspace function IN is used to build the statement matrix AQUERY.

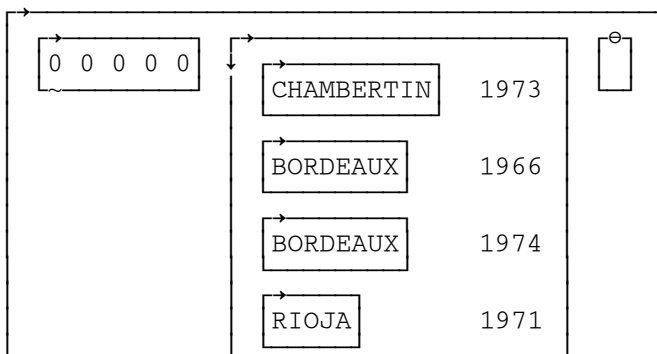
```

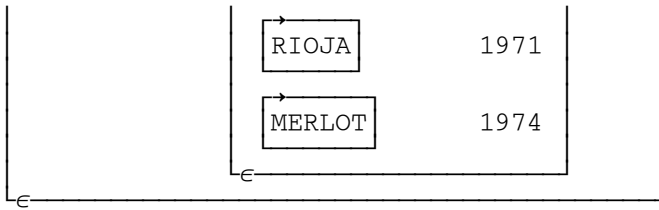
AQUERY←IN
SELECT TYPE, YEAR
FROM STOCKS
WHERE COLOR = 'R'
AND YEAR < 1979

```

With the default option 'MATRIX' option in effect, the result vector looks like this:

```
TAB←SQL AQUERY
DISPLAY TAB
```



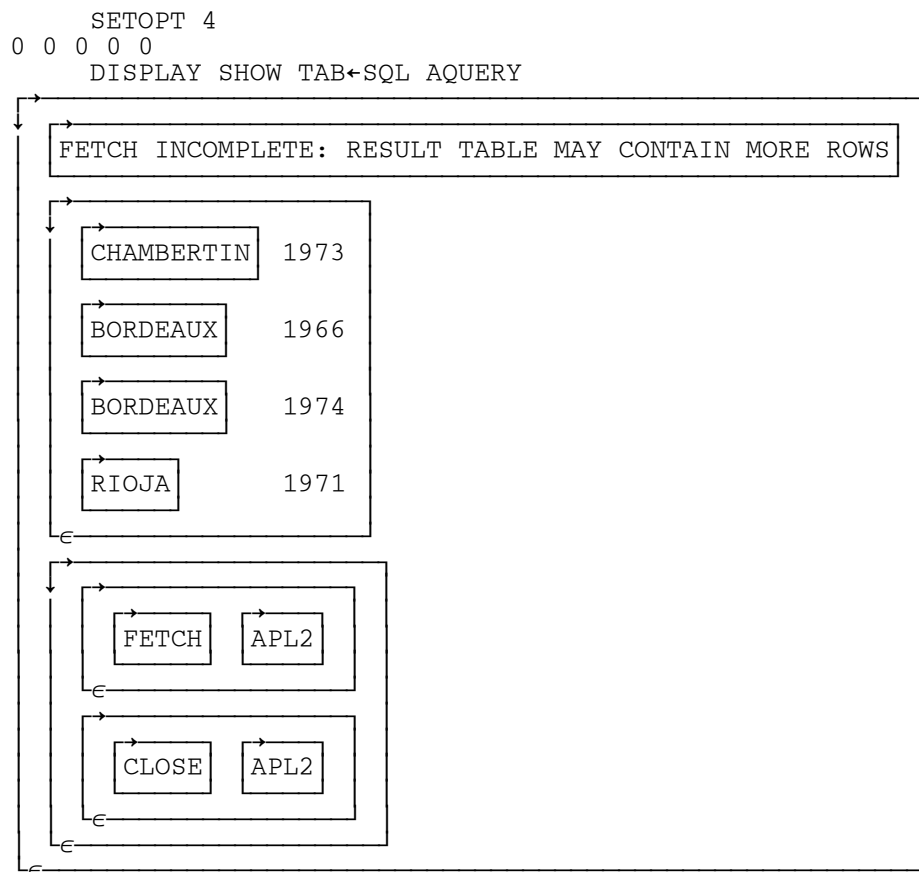


The data item of this result can be retrieved with the *pick* function (I⇒A).

```
DATA←2⇒TAB
DATA
CHAMBERTIN 1973
BORDEAUX    1966
BORDEAUX    1974
RIOJA       1971
RIOJA       1971
MERLOT      1974
ρ'', ''DATA      A LENGTH OF EACH ITEM
10  1
8   1
8   1
5   1
5   1
6   1
```

Stack Vector

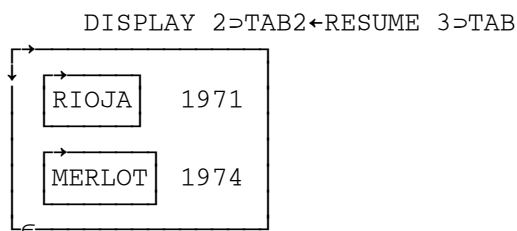
If an SQL function request is not completed, the third item is the remainder of the stack vector.



The SHOW function replaces the return code vector with a message and displays the three items of the result vector down the screen.

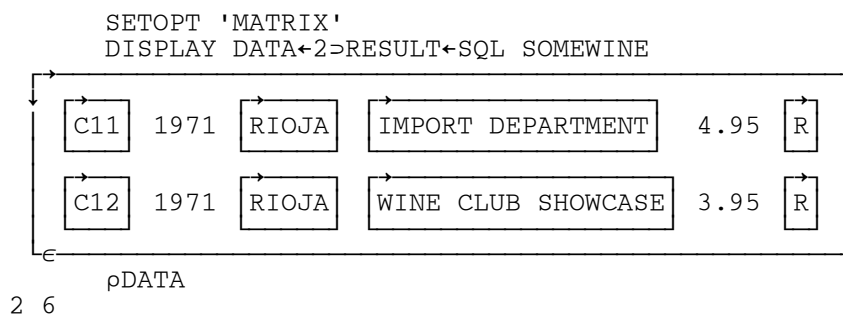
Resuming Execution of the Stack

After the application has processed the data retrieved, the remainder of the stack can be processed by execution of the RESUME function, as in the following example. [User Support Functions under Program Control](#) illustrates a defined function for resuming execution of the stack under program control.



Result of Vector Value-List

The SQL function accepts SQL statements that contain vector-indexes. The value-list passed for the statement can be a simple vector as in the variable SOMEWINE. The 'MATRIX' result of executing SOMEWINE using the SQL function is shown below.



Result of Matrix Argument to SQL Function

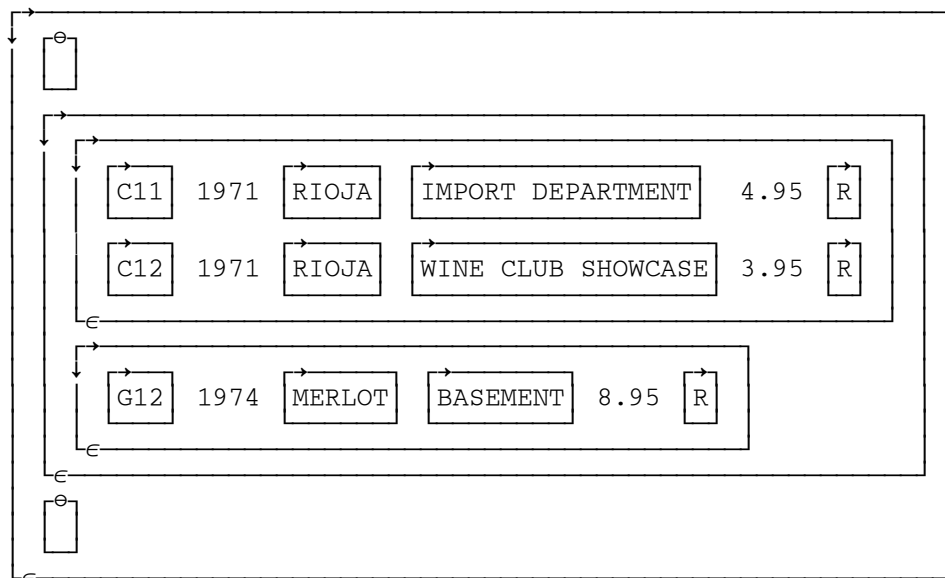
The SQL function also accepts a matrix value-list. The vector-indexes reference each row of the matrix to select the values.

When the SQL function is passed a SELECT statement with a matrix value-list, the result data array is an r by 1 matrix, where r is the number of rows in the value list. The SHOW function can be used to display the result as a stack of vector items. The SHOW function replaces the return code with an empty vector. If the SQL function returns the complete table, the third item of the result vector is an empty vector.

MATRIX Result

In the following example, the SELECTWINES variable is passed as the argument to the SQL function. The SELECTWINES variable contains a 2 by 4 matrix value-list.

```
DISPLAY SHOW RESULT←SQL SELECTWINES
```

```

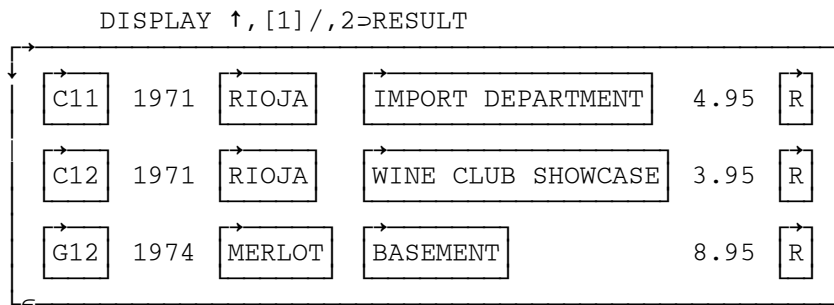
ρSHOW RESULT      A SHAPE OF RESULT OF SHOW FUNCTION
3 1
ρ↑↑RESULT[2]      A SHAPE OF EACH RESULT DATA MATRIX
2 6
1 6

```

The data portion of this result contains two matrixes; one that is 2 by 6 and one that is 1 by 6. The following expression combines these two matrixes to create a single 3 by 6 matrix.

```
↑, [1] / , 2>RESULT
```

The result of applying this expression is displayed below.



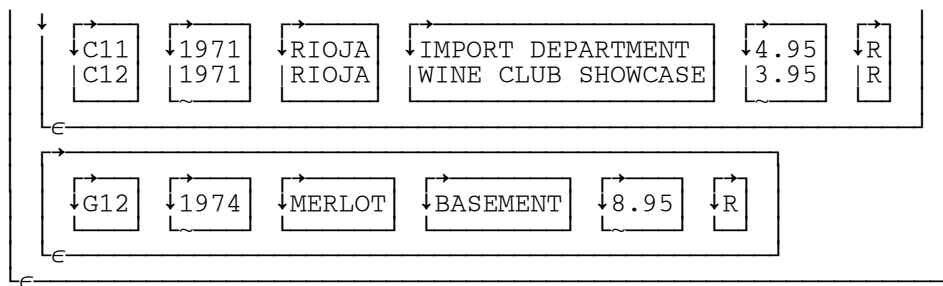
VECTOR Result

In the preceding example, if the 'VECTOR' option were set before the call of the SQL function, the result would be a 2 by 1 matrix containing a vector of simple matrixes in each row.

```

SETOPT 'VECTOR'      A SET THE VECTOR OPTION
0 0 0 0 0
SHOW VRESULT←SQL SELECTWINES  A EXECUTE THE SQL FUNCTION
C11  1971  RIOJA  IMPORT DEPARTMENT  4.95  R
C12  1971  RIOJA  WINE CLUB SHOWCASE  3.95  R
G12  1974  MERLOT BASEMENT            8.95  R
DISPLAY 2>VRESULT

```



User Support Functions under Program Control

The PUTSQL function is an example of a user-written function that updates the data in a row of a table or adds a new row, using values in a variable V. The function depends on two global variables:

- UPD, assigned an SQL UPDATE statement
- INS, assigned an SQL INSERT statement

If the value corresponding to the WHERE clause in the UPDATE statement finds a match in the table, the data in the row is updated. If not, the values in V are added to the end of the table.

```
[0] PUTSQL V;E
[1] A SQL UPDATE - V IS VALUES
[2] A UPD AND INS ARE GLOBAL SQL STATEMENTS
[3] E←↑SQL UPD V           A UPDATE - E IS RETURN CODE
[4] →(E^.=0)/0           A EXIT IF COMPLETED
[5] →((MESSAGE E) ERR ~2>E)/0 A ERROR UNLESS NOT FOUND
[6] E←↑SQL INS V           A INSERT
[7] →((MESSAGE E) ERR Ev.≠0)/0 A RETURN MESSAGE IF ERROR
[0] Z←MSG ERR COND
[1] A PRINT MSG IF COND IS 1
[2] A RETURN COND
[3] →(~Z←COND)/0
[4] MSG
```

Two sample global SQL statements for this function are shown below:

```
UPD
UPDATE STOCKS
SET BIN=:1,YEAR=:2,
STORLOC=:4,COST=:5,
COLOR=:6
WHERE TYPE=:3
INS
INSERT INTO STOCKS
(BIN,YEAR,TYPE,STORLOC,COST,COLOR)
VALUES (:1,:2,:3,:4,:5,:6)
```

The values to be updated or inserted might look like this:

```
VALUES
'X55' 1986 'CHAMPAGNE' 'SPECIALTY CORNER' 15.95 'W'
```

The call of the PUTSQL function would be:

```
PUTSQL VALUES
```

If the type 'CHAMPAGNE' exists, the row is updated. If it does not exist, 'CHAMPAGNE' is added to the table.

Note: To make the updates to the database permanent, and release all table locks (implicit or explicit) you must issue a COMMIT request. The following sequence:

```
PUTSQL VALUES  
COMMIT
```

is analogous to saying:

```
PUT VALUES  
) SAVE
```

where PUT is a function that updates values in the workspace only.

Terminating the Unit of Work

When using AP 127 or AP 227 to communicate with SQL, a *unit of work* is established with the first SQL call. A unit of work (sometimes known as a unit of recovery) defines to SQL a set of changes to the database that is recorded as one logical change.

A unit of work is terminated with a ROLLBACK or COMMIT. A ROLLBACK cancels the unit of work, and a COMMIT makes it permanent.

It is good programming practice to issue COMMIT and ROLLBACK as frequently as possible in order not to tie up resources that might be needed by other users.

There are several ways that the SQL function can be modified to provide responsible termination of the unit of work. One way is to localize the shared variable in the SQL function. When the function terminates, the shared variable is retracted, and at that time the auxiliary processor issues a ROLLBACK.

Another method for ensuring unit of work termination is to provide an *autocommit* facility for the end users. An autocommit facility is one in which the APL2 program issues COMMIT automatically after each logical set of SQL commands that completes without errors.

The example of the UNTIL operator in [Defined Operator UNTIL](#) shows one way to implement an autocommit facility. Another way is to make a second copy of the SQL function, adding a 'COMMIT' to the stack of commands before it is passed to the RESUME function. The users would use the modified copy when they wished to have automatic COMMIT, and the unmodified copy when they did not.

A third way would be to implement a *switch* facility to set the automatic commit on or off, and modify the SQL function to add a 'COMMIT' or 'ROLLBACK' to the stack depending on the value of the switch.

On workstations, some databases support a built-in autocommit facility. Where this is supported, it can be requested when the database connection is made. See [CONNECT](#) for more information.

The SQL Workspace - Reference

This chapter describes the functions and operators in the SQL workspace. For those functions that take arguments, the argument list is either a simple vector (a single item need not be enclosed) or a vector of vectors.

Most results consist of a five-item *return code vector* and a *data array*, as described in [Results](#). When the data array is empty, the syntax shown for the workspace function will retain only the return code.

Some of the examples in the descriptions use the defined function `IN`, which is an auxiliary function in the SQL workspace to build input matrixes. An editor can also be used to build input matrixes. (See *APL2 Programming: System Services Reference* on mainframe systems or the *APL2 User's Guide* on workstation systems for more information on editors.)

- [CALL](#)
- [CHART](#)
- [CHARTDATA](#)
- [CLOSE](#)
- [COMMIT](#)
- [CONNECT](#)
- [DATE](#)
- [DECLARE](#)
- [DESC](#)
- [EVAL](#)
- [EVALSIM](#)
- [EXEC](#)
- [FETCH](#)
- [GETOPT](#)
- [IN](#)
- [ISOL](#)
- [MESSAGE](#)
- [NAMES](#)
- [ODBC](#)
- [ODBCOPEN](#)
- [OFFER](#)
- [OPEN](#)
- [PREP](#)
- [PROCEDURE](#)
- [PURGE](#)
- [PUT](#)
- [QUE](#)
- [QUERY](#)
- [RESUME](#)
- [ROLLBACK](#)
- [SETOPT](#)
- [SHOW](#)
- [SQL](#)
- [SQLCA](#)
- [SQLHELP](#)
- [SQLSTATE](#)

- [SSID](#)
- [STATE](#)
- [STMT](#)
- [TIME](#)
- [TIMESTAMP](#)
- [TRACE](#)
- [UNTIL](#)
- [XMLSIZE](#)

CALL

(CODE *data*) ← CALL *name* [*values*]

Executes a data manipulation (INSERT, UPDATE, DELETE) or stored procedure call SQL statement processed using a previous PREP request.

name

Name of the SQL statement to be called.

values

Vector or matrix of substitution values.

data

For data manipulation statements, unprocessed data from the value-list.

For stored procedure calls, the updated value-list.

For More Information: See auxiliary processor operation [CALL](#).

Examples:

```
      CALL 'D1' ('B10' 1973 'CHAMBERTIN' 'COUNTER')
0 0 0 0 0
      CALL 'D2' 1973
0 0 0 0 0
      CALL 'D3' (<'B10')
0 0 0 0 0
      CALL 'D1' ('B10' (10) 'CHAMBERTIN' 'COUNTER')
0 0 0 0 0
```

CHART

code ← [*values*] CHART *data*

Valid on mainframe systems only.

Creates an SQL SELECT statement based on user input. Passes the results of the select to the Interactive Chart Utility of GDDM, where the user can see the data as a line, bar, surface, or pie chart, a histogram, or a Venn diagram.

values

Used as the selection values in the SQL call. (Optional)

data

Contains the charting information. If empty, the data is prompted for by a call to CHARTDATA.

code

Contains the SQL return code.

Usage Notes:

- CHART fetches the default number of rows (20, or as set by SETOPT), and treats an incomplete fetch as an error.
- If the call to the ICU fails for any reason, CHART issues the SQL workspace error message AP2WSQL16 - CHART CALL HAS FAILED. RETURN CODE IS: ω.
The token ω is the AP 126 return code.
- Most failures are due to an inconsistency in the data. Since many calls work even with inconsistencies, CHART does not attempt to detect them all.

Examples: See [CHARTDATA](#).

CHARTDATA

chartname ← CHARTDATA

Valid on mainframe systems only.

Usage Notes:

- The chart data variable is created by a call to CHARTDATA, which generates the following prompts:

TITLE:

Title of chart

FROM:

Name of SQL table to select from

X-AXES:

Name of column(s) that contain X-data

Y-AXES:

Name of column(s) that contain Y-data

LABELS:

Name of column that contains Y-data descriptors

CONDITIONS:

Trailing clause of SELECT statement

The only required values are FROM and Y-AXIS.

- The ICU call is handled by a call to a subfunction:

ICU *keys labels title x y*

- where *x* and *y* may be lists of values.

Examples:

```
CHART  ''
TITLE:  1985 HOUSEHOLD BUDGET
FROM:   H.BUDGET
X-AXES:
Y-AXES:  INCOME OUTGO
LABELS:  ITEMS
CONDITIONS: WHERE YEAR=1984
```

The above sequence calls SQL with the statement:

```
SELECT INCOME, OUTGO, ITEMS
FROM H.BUDGET
WHERE YEAR=1984
```

It then generates a call to the ICU, passing the specified title, the keys INCOME and OUTGO, the contents of the ITEMS column as labels, and two sets of Y-DATA, from INCOME and OUTGO.

CLOSE

`CODE ← ↑CLOSE name`

Closes an open cursor statement.

name

Name of the cursor statement to be closed.

For More Information: See auxiliary processor operation [CLOSE CURSOR](#)

Examples:

```
      CLOSE 'C1 '  
0 0 0 0 0
```

COMMIT

`CODE ← ↑COMMIT`

Makes permanent all changes made to the database since the last COMMIT or ROLLBACK operation.

For More Information: See auxiliary processor operation [COMMIT WORK](#)

Examples:

```
      COMMIT
0 0 0 0 0
```

CONNECT

On Workstation Systems:

```
(CODE info) ←CONNECT {database | driver} [id password] [ 'AUTOCOMMIT' ]  
(CODE info) ←CONNECT 'RESET '  
(CODE info) ←CONNECT ' '
```

On TSO DB2 Systems:

```
(CODE info) ←CONNECT id password [database]  
(CODE info) ←CONNECT database  
(CODE info) ←CONNECT 'RESET '  
(CODE info) ←CONNECT ' '
```

On CMS SQL/DS Systems:

```
(CODE info) ←CONNECT id password [database]  
(CODE info) ←CONNECT database  
(CODE info) ←CONNECT ' '
```

Specifies the user ID or database server to be accessed, or, if issued with the null parameter, queries the status of the database connection.

'RESET'

On workstation systems, severs the current database connection.

On TSO systems, resets the connection to the default database.

database

Name of the database to be accessed

(CMS - 1 to 18 characters; TSO - 1 to 16 characters; Workstations - not limited by the auxiliary processor. The database and/or operating system may enforce stricter limitations.)

id

User id for database operations

(CMS - 1 to 8 characters; TSO - 1 to 255 characters; Workstations - not limited by the auxiliary processor. The database and/or operating system may enforce stricter limitations.)

password

Password associated with the user id

(CMS - 1 to 8 characters; TSO - 1 to 255 characters; Workstations - not limited by the auxiliary processor. The database and/or operating system may enforce stricter limitations.)

info

Database connection information.

driver

Database connection information string. The syntax of this string is data source-dependent.
'AUTOCOMMIT'

Turns on the database autocommit facility, if any.

For More Information: See auxiliary processor operation [CONNECT](#)

Examples:

A TSO DB2 System:

```
CONNECT ''
0 0 0 0 0 DSN07010
CONNECT 'RESET'
0 0 0 0 0 DSN07010
```

A CMS SQL/DS System:

```
CONNECT 'MYUSERID' 'MYPASSWD'
0 0 0 0 0 ARI03030 MYUSERID SQLDBA
CONNECT 'MYUSERID' 'MYPASSWD' 'MYDATABS'
0 0 0 0 0 ARI03030 MYUSERID MYDATABS
```

A Workstation System:

```
CONNECT 'DSN=APL2TEST' 'AUTOCOMMIT'
0 0 0 0 0 DSN=APL2TEST;DBALIAS=APL2TEST
CONNECT 'DBQ=Northwind.mdb;DRIVER={Microsoft Access Driver (*.mdb)}'
0 0 0 0 0 DBQ=Northwind.mdb;DefaultDir=.;Driver={Microsoft Access Driver
(*.mdb)};DriverId=281;FIL=MS Access;MaxBufferSize=2048;PageTimeout=5;
```

DATE

`DATE ← [format] DATE ts`

Formats a numeric time vector into SQL character string DATE format.

format

Desired format (0=ISO, 1=USA, 2=EUR, and 3=JIS)

format can be omitted. If the variable `TFMT` exists, the date is formatted according to `TFMT`. If the variable `TFMT` does not exist, the ISO format is used.

If *format* (or `TFMT`) contains an invalid argument, then the ISO format is used.

ts

□TS or a similar numeric vector.

Examples:

```
DATE □TS
1995-05-09
2 DATE □TS
09.05.1995
DATE 1986 7 9
1986-07-09
```

DECLARE

`CODE ← ↑DECLARE name ['HOLD' | 'NOHOLD']`

Defines a statement name and optionally assigns the HOLD attribute to it. SELECT statements with the HOLD attribute retain their cursor positions across COMMIT.

name

Name of the cursor statement to be declared.

'HOLD'

Specifies that the HOLD attribute should be applied to this statement name.

'NOHOLD'

Specifies that the HOLD attribute should not be applied to this statement name. (This is the default.)

For More Information: See auxiliary processor operation [DECLARE](#)

Examples:

```
      DECLARE 'C1' 'HOLD'
0 0 0 0 0
      DECLARE 'C2'
0 0 0 0 0
```


DESC

`(CODE data) ← DESC name [type]`

Returns result table column information.

name

Name of a SELECT statement that has been processed using a PREP or ODBCOPEN request, or name of a stored procedure call statement that has been processed by a CALL request followed by an OPEN request.

type

One of the following:

'NAMES ' (return column names only)

'LABELS ' (return column labels only)

'BOTH ' (return both names and labels)

'ANY ' (return labels if they exist, otherwise return names)

If no type is specified, 'NAMES ' is the default.

data

Nested matrix containing column information.

For More Information: See auxiliary processor operation [DESCRIBE](#)

Examples:

	DESC	'C1'				
0	0	0	0	0		
		BIN	YEAR	TYPE	STORLOC	COST
		C 3	S	V 12	V 20	D 7 2
		NOT NULL	NULL	NOT NULL	NULL	NULL

EVAL

table ← EVAL *matrix*

Evaluates a matrix that contains mixed character and numeric data.

matrix

Matrix that contains a combination of character strings in quotes and unquoted numeric digits.

table

Nested matrix of mixed character and numeric data.

Usage Notes:

- EVAL can be used with the IN function, or to convert objects created by an editor.
- Character data must be included in quotes for this function to evaluate properly.
- Use ' ' or (⋈0) to indicate a null value.

Examples:

```
DATA←EVAL IN
1 'FRENCH HEN' 'MONDAY'
2 'CALLING BIRDS' 'TUESDAY'
3 'TURTLE DOVE' 'WEDNESDAY'
DISPLAY DATA
```

1	FRENCH HEN	MONDAY
2	CALLING BIRDS	TUESDAY
3	TURTLE DOVE	WEDNESDAY

EVALSIM

table ← *selectnum* *lengths* EVALSIM *matrix*

Evaluates a mixed matrix using descriptors of lengths and types.

matrix

Simple character matrix.

lengths

Vector of column lengths.

selectnum

Indicator of which columns are numeric.

table

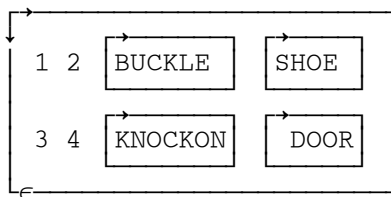
Nested matrix of mixed character and numeric data.

Usage Notes:

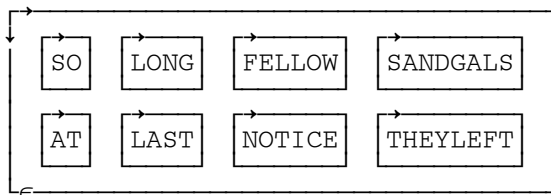
- It is sometimes useful to read simple flat file character data into a workspace and create SQL tables from it. EVALSIM partitions the columns of a simple matrix according to the lengths provided and produces an SQL matrix by evaluating the columns to numbers or characters.
- EVALSIM provides a user-modifiable error exit function, EXIT, whose default action is to provide terminal output of the error. If there is an evaluation error, a message is printed, and the attendant data is not evaluated.

Examples:

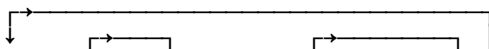
```
MM←⌵' 1 2BUCKLE SHOE ' ' 3 4KNOCKON DOOR'
B←1 1 0 0
L←2 2 7 5
DISPLAY B L EVALSIM MM
```

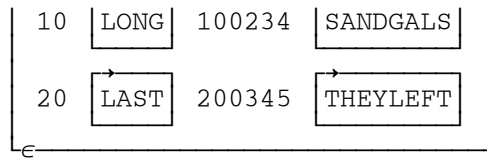


```
B←1 0 1 0
L←2 4 6 8
MM←⌵'SO LONG FELLOW SANDGALS' 'AT LAST NOTICE THEYLEFT'
DISPLAY B L EVALSIM MM
```



```
MM←⌵'10LONG100234SANDGALS' '20LAST200345THEYLEFT'
DISPLAY B L EVALSIM MM
```

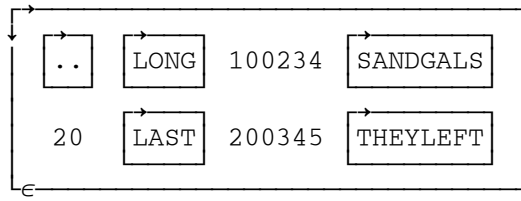




MM<=>'..LONG100234SANDGALS' '20LAST200345THEYLEFT'

DISPLAY B L EVALSIM MM

SYNTAX ERROR : ..



EXEC

CODE ← ↑EXEC *stmt*

Executes an SQL statement without parameters.

stmt

Any SQL statement except SELECT, CONNECT, COMMIT, and ROLLBACK.

For More Information: See auxiliary processor operation [EXECUTE](#)

Examples:

```
EXEC IN
CREATE TABLE STOCKS
(BIN CHAR(3) NOT NULL,
YEAR SMALLINT,
TYPE VARCHAR(12) NOT NULL,
STORLOC VARCHAR(20),
COST DECIMAL(6,2),
COLOR CHAR(1))
0 0 0 0 0
EXEC IN
INSERT INTO STOCKS
(BIN, YEAR, TYPE, STORLOC)
VALUES ('B10', 1973, 'CHAMBERTIN', 'COUNTER')
0 0 0 0 0
```

FETCH

`(CODE data) ← FETCH name [options..]`

Returns new result table data as the second item of the shared variable result vector.

name

Name of an open cursor statement.

options

Any one or all of the following items in any order:

- A number that specifies the maximum number of rows to be retrieved.
- A character vector indicating the data transfer option 'MATRIX' or 'VECTOR'.
- A character vector indicating the length matrix option 'LENGTH' or 'NOLENGTH'.

data

SQL table rows. The data is returned as either a matrix or a vector of simple matrices, depending on the data option chosen.

For More Information: See auxiliary processor operation [FETCH](#)

Examples:

```
      FETCH 'C1' 'VECTOR' 100
0 0 0 0 0  B10  1973  CHAMBERTIN  COUNTER      10.95  R
           B11  1966  BORDEAUX    SPECIALTY COUNTER  8.95  R
           B12  1974  BORDEAUX    SPECIALTY COUNTER  10.95 R
           C11  1971  RIOJA       IMPORT DEPARTMENT  4.95  R
           C12  1971  RIOJA       WINE CLUB SHOWCASE  3.95  R
           F16  1983  ROSE        WINDOW             0
           G12  1974  MERLOT      BASEMENT           8.95  R
           I10  1979  BARDOLINO   ANNEX              2.25  R
           I11  1979  VALPOLICELLA ANNEX             2.25  R
           K10  1981  CHABLIS     ON ORDER           3.95  W
           K11  1981  RIESLING    SHELF              6.25  W
```

GETOPT

(CODE *options*) ← GETOPT

Returns the current settings of the options-list.

options

Three-item vector containing the current settings of the auxiliary processor options:

1. 'MATRIX' or 'VECTOR' (result data format)
2. 'LENGTH' or 'NOLENGTH' (length matrix indicator)
3. Number of rows to be returned from the result table.

For More Information: See auxiliary processor operation [GET OPTIONS](#)

Examples:

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

IN

$data \leftarrow IN$

Accepts input until an empty line is entered. The result is a character matrix containing the input lines.

Examples:

```
MAT←IN
FIRST INPUT LINE
MIDDLE INPUT LINE
LAST INPUT LINE
```

blank line is end of input

```
MAT
FIRST INPUT LINE
MIDDLE INPUT LINE
LAST INPUT LINE
```

result is a matrix

ISOL

```
(CODE setting) ← ISOL ''
```

Queries the isolation level.

```
CODE ← ↑ISOL setting
```

Sets the isolation level.

setting

One of the following values:

- 'CS' for *Cursor Stability* (ODBC: *Read Committed*)
- 'RR' for *Repeatable Read* (ODBC: *Serializable*)
- 'RS' for *Read Stability* (ODBC: *Repeatable Read*)
- 'UR' for *Uncommitted Read* (ODBC: *Read Uncommitted*)

For More Information: See auxiliary processor operations [SET ISOLATION LEVEL](#) and [GET ISOLATION LEVEL](#)

Examples:

```
      ISOL 'RR'
0 0 0 0 0
      ISOL ''
0 0 0 0 0  RR
```

MESSAGE

`MSG ← MESSAGE rcode`

Returns a message associated with a return code from the auxiliary processor or from a workspace function.

*r*code

The five-item code returned from the auxiliary processor or an SQL workspace function at the time of failure or request completion.

This code must be one of the following:

- A code indicating a message from the auxiliary processor. MESSAGE returns information about the most recent occurrence of the message.
- A code indicating a message from the SQL database manager. MESSAGE returns information about the most recent call to SQL.
- An error code from a workspace function.

Usage Notes:

- The MESSAGE function is similar to the auxiliary processor operation 'MSG'. The MESSAGE function, however, can be used to return messages based on error codes from workspace functions, while 'MSG' returns messages only from the auxiliary processor and the SQL database manager.
- If no error has occurred (the return code passed is 0 0 0 0 0), a null character vector is returned.
- MESSAGE is invoked by the user support function SHOW, but it can also be invoked directly.

For More Information: See auxiliary processor operation [GET MESSAGE](#)

Examples:

```
MESSAGE 1 0 0 1 114
ERROR MESSAGE.
SQL statement not character vector or matrix.
MESSAGE ↑CALL 'D2' 1973
MESSAGE 1 0 0 2 ^934
TRANSACTION BACKOUT.
^934
ARICMOD  NUCXLOAD
ARIRVSTC
28 0 0 0 0 0
S      S
58021
```

NAMES

`(CODE names) ← NAMES`

Returns a list of active statement names.

names

Vector of character vectors containing the names of all active statements.

For More Information: See auxiliary processor operation [GET NAMES](#)

Examples:

```
      NAMES
0 0 0 0 0  C1 C2
```

ODBC

(CODE *info*) ← ODBC *type*

Returns ODBC system information.

Note: This command is valid for AP 227 only.

type Character vector indicating which type of system information should be returned.

'DATASOURCES ' to request the output from the ODBC service *SQLDataSources*. This service returns a list of the available ODBC Data Sources (databases) and their associated drivers.

'DRIVERS ' to request the output from the ODBC service *SQLDrivers*. This service returns a list of the installed ODBC Drivers and their attributes.

info The requested information.

For More Information: See auxiliary processor operation [ODBC INFORMATION](#)

Examples:

```
      ODBC 'DATASOURCES '  
0 0 0 0 0      APL2TEST IBM DB2 ODBC DRIVER  
              MQIS      SQL Server
```

ODBCOPEN

CODE ← ↑ODBCOPEN *name type*

Sets up an SQL table containing information about the currently connected ODBC database. Rows of the table can be retrieved by subsequent FETCH requests in the same manner as for cursors created by the PREP and OPEN requests.

Note: This command is valid for AP 227 only.

name

Identifies the name to be used for the cursor.

type

One of the following keywords:

'GETTYPEINFO' to request the output from the ODBC service *SQLGetTypeInfo*. This service builds a table with one row for each SQL data type supported by the currently connected data source. Each row contains the data type name, code, and other information about how it is defined. The number and content of the columns may vary between data sources.

'TABLES' to request the output from the ODBC service *SQLTables*. This service builds a table with one row for each SQL table, view, and schema defined in the currently connected data source. Each row contains the table name, type, and other information about how it is defined. The number and content of the columns may vary between data sources.

For More Information: See auxiliary processor operation [ODBC OPEN](#)

Examples:

```
      ODBCOPEN 'NAME' 'TABLES'
0 0 0 0 0
      FETCH 'NAME' 5
0 0 1 0 0      SYSIBM SYSATTRIBUTES      SYSTEM TABLE
                SYSIBM SYSBUFFERPOOLNODES  SYSTEM TABLE
                SYSIBM SYSBUFFERPOOLS       SYSTEM TABLE
                SYSIBM SYSCHECKS            SYSTEM TABLE
                SYSIBM SYSCOLAUTH           SYSTEM TABLE
```

OFFER

CODE ← OFFER

Executes \square SVO for a variable named DAT with the auxiliary processor number in variable SQL_AP and sets the access control vector to 1 0 1 1.

For a description of \square SVO, see *APL2 Programming: Language Reference*.

For more about the access control vector, see [Shared Variable Requirements](#).

OPEN

`CODE ← ↑OPEN name [values]`

Opens a previously prepared SQL statement.

name

Name of a previously prepared SQL (SELECT or, on CMS, INSERT) statement, or the name of a stored procedure call processed by PREP and CALL.

values

Vector of values to be selected by vector-indexes passed as part of the prepared SQL statement, if a SELECT. (For an INSERT statement, the value-list is passed at the PUT step. For a stored procedure call, the value-list is passed at the CALL step.)

For More Information: See auxiliary processor operation [OPEN CURSOR](#)

Examples:

```
      OPEN  'C1 '
0 0 0 0 0
      OPEN  'C2 '  (= 'M75 ')
0 0 0 0 0
      OPEN  'C5 ' 15000
0 0 0 0 0
```

PREP

`CODE ← ↑PREP name stmt`

Prepares an SQL statement for later execution by a CALL or OPEN request.

name

Identifies the SQL statement to be prepared.

stmt

Any SELECT, INSERT, UPDATE, DELETE or stored procedure call statement can be prepared. The statement may or may not contain APL2 vector-indexes.

For More Information: See auxiliary processor operation [PREPARE](#)

Examples:

```
      PREP 'D1' IN
INSERT INTO STOCKS
(BIN, YEAR, TYPE, STORLOC)
VALUES (:1, :2, :3, :4)
0 0 0 0 0
      PREP 'D2' IN
UPDATE STOCKS
SET YEAR = :1
WHERE BIN = 'B10'
0 0 0 0 0
      PREP 'D3' IN
DELETE FROM STOCKS
WHERE BIN = :1
0 0 0 0 0
      PREP 'S1' IN
SELECT NAME, SALARY
FROM PERSONNEL
WHERE DEPT = 'M75'
0 0 0 0 0
      PREP 'C1' IN
CALL PROUTINE (:1, :2, :3)
0 0 0 0 0
```


PROCEDURE

`TABLES ← PROCEDURE stmt [values]`

Calls the SQL function to process a stored procedure call, then fetches all available result sets.

Usage Notes:

- PROCEDURE returns a 1-column matrix. The first row of the matrix contains the modified parameters passed in *values*, or null if no values were passed. The remaining rows contain the result sets created by the procedure, if any.
- Each result set returned is formatted in the same manner as tables fetched by [QUERY](#) using SELECT statements.
- PROCEDURE calls the user-modifiable functions REPORT, HEAD, and APPEND. See [QUERY](#) for more information on these functions.

Examples:

```
PROCEDURE 'CALL TESTPROC(:1, :2, :3)' ('      ' 1979 'W')
00000 1979 W
BIN YEAR TYPE          STORLOC      COST  COLOR
I10 1979 BARDOLINO     ANNEX        2.25  R
I11 1979 VALPOLICELLA ANNEX        2.25  R
BIN YEAR TYPE          STORLOC      COST  COLOR
K10 1981 CHABLIS       ON ORDER    3.95  W
K11 1981 RIESLING      SHELF       6.25  W
```

PURGE

`CODE ← ↑PURGE name`

Removes one statement or all statements from the list of active names.

name

Name of statement to be removed, or ' ' (empty character vector) to remove all active statement names.

For More Information: See auxiliary processor operation [PURGE](#)

Examples:

```
      PURGE 'C1 '  
0 0 0 0 0  
      PURGE ''  
0 0 0 0 0
```

PUT

(CODE *data*) ← PUT *name values*

Valid on CMS systems only.

PUT executes an INSERT statement processed by previous PREP and OPEN requests.

name

Name of the SQL statement to be called.

values

Vector or matrix of substitution values. The vector, or each row of the matrix, is indexed using the index values provided during the PREP, the value is passed to SQL for substitution into the statement. SQL is called once for a vector or each row of a matrix.

data

Unprocessed data from the value-list. If all data was processed successfully, this item is a simple character null.

For More Information: See auxiliary processor operation [PUT](#)

Examples:

```
      PUT 'D1' ('B10' 1973 'CHAMBERTIN' 'COUNTER')
0 0 0 0 0
      PUT 'D1' ('B10' (10) 'CHAMBERTIN' 'COUNTER')
0 0 0 0 0
```

QUE

(CODE *data ustack*) ← QUE *stack*

Calls the auxiliary processor iteratively to process a stack vector of one or more operation requests.

stack

Vector of auxiliary processor operation requests. In each item of the vector, the operation request is followed by arguments particular to the operation.

data

A result data array for each successfully processed operation in the stack. Any of these items can be an empty vector.

ustack

Any unprocessed items from *stack*. If all operations process successfully, this item is empty.

Usage Notes:

- CODE is the five-item return code associated with the *last* operation processed.
- The stacked operations are *sequentially dependent*. When an operation causes an error, processing is terminated, and the incomplete result is returned to the user along with the remaining operations stack.
- The QUE function does not terminate processing if the five-item APL2 return code is a member of the global variable CODE_. This provides the ability to bypass expected messages.
- The QUE function is normally called by the RESUME function after SQL has created a stack of operations or when a table has not been completely retrieved (see [SQL](#).)

QUERY

```
TABLE ← QUERY stmt [values]
```

Calls the SQL function with a SELECT statement until all requested table rows are processed.

Usage Notes:

- QUERY calls the user-modifiable functions REPORT, HEAD, COMBINE, and APPEND.
- QUERY normally returns a single table that contains the heading appended to all fetched tables, as modified by REPORT. The user can change this by replacing the APPEND function.

Subfunctions of QUERY:

```
report ← head REPORT table
```

REPORT is a dummy function that can be altered or replaced to do operations on a table.

```
report ← HEAD table
```

Appends a description to *table*, using DESC.

```
table ← stmt COMBINE data
```

Combines the results of a fetch with a matrix *values* list into a single matrix.

```
table ← APPEND table
```

APPEND appends two tables. It is separately defined so that the user who does not desire such an explicit result may alter or replace it.

Examples:

```
      SETOPT 5
0 0 0 0 0
      QUERY'SELECT * FROM STOCKS'
BIN YEAR TYPE      STORLOC      COST  COLOR
B10 1973 CHAMBERTIN COUNTER    10.95  R
B11 1966 BORDEAUX   SPECIALTY COUNTER    8.95  R
B12 1974 BORDEAUX   SPECIALTY COUNTER    10.95  R
```

C11	1971	RIOJA	IMPORT DEPARTMENT	4.95	R
C12	1971	RIOJA	WINE CLUB SHOWCASE	3.95	R
F16	1983	ROSE	WINDOW		
G12	1974	MERLOT	BASEMENT	8.95	R
I10	1979	BARDOLINO	ANNEX	2.25	R
I11	1979	VALPOLICELLA	ANNEX	2.25	R
K10	1981	CHABLIS	ON ORDER	3.95	W
K11	1981	RIESLING	SHELF	6.25	W

RESUME

$(\text{CODE } data \text{ ustack}) \leftarrow \text{RESUME } stack$

Calls the QUE function to process a stack vector of one or more operation requests.

stack

Vector of auxiliary processor operation requests. In each item of the vector, the operation request is followed by arguments particular to the operation.

data

FETCH result data. If multiple FETCH requests are issued, this item is a one-column matrix of result data arrays. It can be an empty vector.

ustack

Any unprocessed items from *stack*. If all operations process successfully, this item is empty.

Usage Notes:

- CODE is the five-item return code associated with the *last* operation processed.
- RESUME is called by the SQL function, which prepares a request stack.
- RESUME can also be used to resume incomplete processing. For example, it can be used when more rows of a result table remain to be retrieved using a FETCH request.

Examples:

To process Remaining Stack:

```
RESULT ← SQL QUERY1  
RESUME 3>RESULT
```

ROLLBACK

`CODE ← ↑ROLLBACK`

Backs out all changes made to the database since the last COMMIT or ROLLBACK request. The state of all prepared statements is reset to unprepared. All locks are released.

For More Information: See auxiliary processor operation [ROLLBACK WORK](#)

Examples:

```
      ROLLBACK
0 0 0 0 0
```


SETOPT

`CODE ← ↑SETOPT options...`

Sets the values of the auxiliary processor options.

options

Any one or all of the following items in any order:

- A number that specifies the maximum number of rows to be retrieved.
- A character vector indicating the data transfer option 'MATRIX' or 'VECTOR'.
- A character vector indicating the length matrix option 'LENGTH' or 'NOLENGTH'.

For More Information: See auxiliary processor operation [SET OPTIONS](#)

Examples:

```
      SETOPT 'VECTOR' 'LENGTH' 35
0 0 0 0 0
```

SHOW

`RESULT ← SHOW sqlresult`

Displays error messages and reshapes the three-item SQL function result vector so that its nested components display as stacked vectors or matrixes.

sqlresult

The result from a call to the SQL function.

Usage Notes:

- SHOW calls the MESSAGE function to obtain error messages. MESSAGE displays a completion code description such as TRANSACTION BACKOUT along with an error message in place of the return code. It also replaces an all-zero return code with an empty vector. Thus, just the data vector is seen on a normal return from a FETCH request.

Examples:

```
      SHOW SQL IN
SELECT YEAR, TYPE, STORLOC
FROM STOCKS
WHERE COST > 6.00
B10 1973 CHAMBERTIN    COUNTER      10.95  R
B11 1966 BORDEAUX      SPECIALTY COUNTER   8.95  R
B12 1974 BORDEAUX      SPECIALTY COUNTER  10.95  R
G12 1974 MERLOT        BASEMENT      8.95  R
K11 1981 RIESLING      SHELF         6.25  W
```

SQL

(CODE *data ustack*) ← SQL *stmt* [*values*]

Allows you to process SQL language statements without specifying auxiliary processor operation codes. SQL examines the argument vector and creates the appropriate operations stack. It then passes the operations stack to the RESUME function for processing.

stmt

Any SQL statement.

If the first word of *stmt* is COMMIT or ROLLBACK, the statement is added to the stack.

If the first word is SELECT or WITH, the request stack for a cursor statement is formed: 'PREP' 'OPEN' 'FETCH' 'CLOSE'.

If *stmt* begins with CALL, a 'PREP' 'CALL' request stack is formed.

If *stmt* begins with INSERT, UPDATE, or DELETE and a *value-list* is present, the statement is assumed to be a defined statement, and a 'PREP' 'CALL' request stack is formed.

If *stmt* is GETTYPEINFO or TABLES, a cursor statement is formed with 'ODBCOPEN' 'FETCH' 'CLOSE'. (AP 227 only)

Otherwise, *stmt* is assumed to be an immediate statement, and an 'EXEC' request is formed.

For a definition of cursor, defined, and immediate statements, see [Processing SQL Statements under APL2](#).

values

Vector or matrix of values to be selected by vector-indexes specified in the SQL statement.

If *values* is a matrix and the statement is SELECT, the 'OPEN' 'FETCH' 'CLOSE' operations sequence is repeated for each row of the matrix. This facility can be used to accumulate the results of an SQL SELECT statement that has more than one set of values for its search conditions.

data

Result data, if any.

ustack

Any unprocessed items from the stack of operations created. If all operations process successfully, this item is empty.

Usage Notes:

- When creating a 'FETCH' operation, the SQL function does *not* pass any options-list items. You can use the SETOPT function to change the options-list values.

- For cursor and data manipulation statements, the SQL function always uses the statement name APL2. The statement named APL2 always retains its latest status upon exit from the SQL function, whether or not the entire operations stack has been processed successfully.

Auto Commit:

To automatically commit after a call to the SQL function, call with the statement:

```
SQL UNTIL args 'COMMIT'
```

where *args* is the list of SQL function arguments.

Error Processing:

As shipped by IBM, the SQL function suspends processing the commands it has stacked whenever a return code is received that is not (0 0 0 0 0). Under certain conditions, warning return codes such as (0 1 0 2 0) can be received. For example, on CMS systems a warning condition is set when it must suspend blocking due to LONG VARCHAR data. You may want the SQL function to continue processing for these cases.

The variable CODE_ in the SQL workspace controls the actions of the SQL function during return code processing. As shipped, it contains only the value (0 0 0 0 0) in its list of valid return codes. You can modify this variable, adding to it any other return codes you want considered as nonerror return codes.

Examples:

```
SELECTWINES←IN
SELECT * FROM STOCKS
WHERE TYPE = :1
AND YEAR = :2
VALUES←⇒('RIOJA' 1971)('MERLOT' 1974)
SQL SELECTWINES VALUES
0 0 0 0 0      C11 1971 RIOJA IMPORT DEPARTMENT 4.95 R
                C12 1971 RIOJA WINE CLUB SHOWCASE 3.95 R
                G12 1974 MERLOT BASEMENT 8.95 R
```

SQLCA

`(CODE sqlca) ← SQLCA`

Retrieves the current contents of the SQL communications area (SQLCA).

After an SQL operation, this control block contains return codes and, if an error has occurred, additional diagnosis information.

sqlca

A 6 by 1 matrix containing the following six items:

1. SQLCODE (integer)
2. SQLERRM message tokens
3. SQLERRP routine name
4. SQLERRD return codes (a six-item vector)
5. SQLWARN, an 11-character vector of warning indicators
6. SQLSTATE, a 5-character vector

For a definition of these fields, see the *SQL Reference* for your database system.

For More Information: See auxiliary processor operation [GET SQLCA](#)

Examples:

```
      SQLCA
0 0 0 0 0  -934
      ARICMOD  NUCXLOAD
      ARIRVSTC
      28 0 0 0 0 0
      S      S
      58021
```

SQLHELP

DATA ← SQLHELP keyword

Valid on CMS systems only.

Retrieves help text from database tables.

keyword

Character vector containing a string to be used to search the table. For example, any negative SQLCODE (such as ' -204 ') or any SQL language keyword (such as ' SELECT ').

Usage Notes:

- If no help text exists for the keyword, the result data is empty.
- SQL does not recognize the APL2 negative sign (⌵.) When passing negative SQLCODEs the regular minus (-) must be used.

Examples:

```
SQLHELP 'UPDATE'  
SQLHELP '-934'
```

SQLSTATE

(CODE *sqlstate*) ← SQLSTATE

Retrieves the current contents of the SQLSTATE variable. SQLSTATE is a status indicator that is common on all IBM relational database systems.

sqlstate

A 5-element character vector containing the current contents of SQLSTATE.

For More Information: See auxiliary processor operation [GET SQLSTATE](#)

Examples:

```
      SQLSTATE
0 0 0 0 0 58021
```

SSID

```
(CODE setting) ← SSID ' '
```

Queries the subsystem ID.

```
CODE ← ↑SSID setting
```

Sets the subsystem ID.

Valid on TSO systems only.

setting

Character vector containing the subsystem ID.

For More Information: See auxiliary processor operations [SET SSID](#) and [GET SSID](#)

Examples:

```
      SSID 'DSN2 '  
0 0 0 0 0  
      SSID ' '  
0 0 0 0 0   DSN2
```


STATE

$(\text{CODE } state) \leftarrow \text{STATE } name$

Yields the current state for an SQL statement.

name

Name of the SQL statement for which the state is desired.

state

Two-item nested vector. The first item is the two-element numeric state vector as for operation 'STATE'. The second item is a character string translation of the numeric state vector.

For More Information: See auxiliary processor operation [GET STATE](#)

Examples:

```
STATE 'C1'
0 0 0 0 0 2 2  SELECT OPEN
```

STMT

(CODE *stmt*) ← STMT *name*

Yields a specified SQL statement.

name

Name of the SQL statement to be returned.

stmt

Value of the SQL statement as originally passed on the PREP request.

For More Information: See auxiliary processor operation [GET STMT](#)

Examples:

```
      STMT 'C1'
0 0 0 0 0  SELECT * FROM MYTABLE WHERE COLOR = 'YELLOW'
```

TIME

`TIME ← [format] TIME ts`

Formats a numeric time vector into SQL character string TIME format.

format

Desired format (0=ISO, 1=USA, 2=EUR, and 3=JIS)

format can be omitted. If the variable TFMT exists, the date is formatted according to TFMT. If the variable TFMT does not exist, the ISO format is used.

If *format* (or TFMT) contains an invalid argument, then the ISO format is used.

ts

□TS or a similar numeric vector.

Examples:

```
TIME □TS
18.35.06
3 TIME □TS
18:35:06
TIME 1995 5 9 7 25 52
07.25.52
```

TIMESTAMP

`TIMESTAMP ← TIMESTAMP ts`

Formats a numeric time vector into SQL character string TIMESTAMP format.

ts

□TS or a similar numeric vector.

Usage Notes:

- TIMESTAMP always produces ISO format.

Examples:

```
TIMESTAMP □TS
1995-05-09-18.37.03.493
TIMESTAMP 1986 7 9 7 25 52
1986-07-09-07.25.52.000
```

TRACE

```
(CODE level...) ← TRACE ''
```

Returns the current trace settings.

```
CODE ← ↑TRACE (modnum level) ...
```

Sets tracing levels.

modnum

Module number for tracing. Module number 0 specifies all modules. Other module numbers vary by system.

level

Trace level number. Trace level 0 specifies no tracing. Trace level 9 specifies the maximum level of tracing.

For More Information: See auxiliary processor operations [GET TRACE](#) and [SET TRACE](#)

Examples:

```
TRACE (1 5)
0 0 0 0 0
TRACE (0 3) (4 9)
0 0 0 0 0
TRACE ''
0 0 0 0 0 3 3 3 9 3 3 3 3 3 3
```

UNTIL

$results \leftarrow (f \text{ UNTIL}) stack$

The UNTIL operator applies the function f to each *stack* item *until* an error occurs or execution is completed.

stack

Vector of auxiliary processor operation requests. In each item of the vector, the operation request is followed by arguments particular to the operation.

results

Vector of result vectors from the operation requests

Usage Notes:

- f must be a function that returns an error code vector as its first item. A nonzero item in the error code vector signals an error.
- UNTIL is similar to the primitive operator *each* (`¨`), except that UNTIL considers the error code of one application of f before processing the next.

Examples:

```
SQL UNTIL IN 'COMMIT'
INSERT INTO STOCKS
(BIN, YEAR, TYPE, STORLOC, COST, COLOR)
VALUES ('I10', 1979, 'BARDOLINO', 'ANNEX', 2.25, 'R')
0 0 0 0 0    0 0 0 0 0
```

Note: This call of UNTIL illustrates the importance of issuing a COMMIT to make permanent all changes to the database.

XMLSIZE

```
(CODE size) ← XMLSIZE ''
```

Returns the current size for fetching XML data items.

```
CODE ← ↑XMLSIZE size
```

Sets the size to be used for fetching XML data items.

size

The number of bytes of storage to be allocated for fetching each XML data item. XML data items longer than *size* will be truncated.

If no size is specified by the application, the default size is 65536 bytes.

For More Information: See auxiliary processor operations [GET XML SIZE](#) and [SET XML SIZE](#)

Examples:

```
XMLSIZE ''
0 0 0 0 0 65536
XMLSIZE 128000
0 0 0 0 0
XMLSIZE ''
0 0 0 0 0 128000
```

Special Programming Techniques

This section contains the following topics:

- [Concurrent Update of Shared Tables](#)
- [Interrupting SQL Processing](#)
- [Handling the LOB Datatypes](#)
- [Support for DBCS Characters](#)

Concurrent Update of Shared Tables

Updating records in a shared database must be undertaken with care. Consider the following scenario:

1. User 1 SELECTs fields A,B,C, and D from a certain database record into an APL workspace for analysis.
2. User 2 SELECTs the same fields from the same record.
3. User 2 completes analysis and issues the SQL UPDATE, updating fields C and D.
4. User 1 completes analysis and issues the SQL UPDATE, modifying fields B and D.
5. When User 1 updated the record, the update of field D overlaid the data that User 2 had updated. User 2's update of field D was lost.

This classic problem is called the concurrent update problem. When you take records out of the database, and put them under control of APL2, you must take responsibility for ensuring that you do not undo another user's data. This appendix discusses the means for ensuring the integrity of data in a cooperative SQL/APL2 shared data application environment.

Various methods are available for handling concurrent update of shared data. Their use requires an understanding of:

- [Authorization](#)
- [Isolation Levels](#)
- [Fully-Qualified Update](#)
- [Table Locking](#)
- [Deadlock Handling](#)

The following sections contain discussions of these concepts and their usage. They should be treated as an overview rather than a guide. For comprehensive descriptions particular to each database system, you should read the discussions on concurrent access and isolation levels in the documentation for your database system.

Authorization

The database systems have authorization schemes that prevent unauthorized users from accessing data. You have to be authorized, either implicitly or explicitly, in order to have an APL program, or any other program, retrieve or update data.

The simplest technique to prevent concurrent update is to authorize only one user to update a given field. If one, and only one, person is given the authority to update (for example) a SALARY field in the EMPLOYEE table, the SQL system ensures that no one else can update that field in that table. Such authorization can be controlled with the GRANT/REVOKE commands. Note, however, that if several TSO users share the same authorization ID, then they may update the same field concurrently.

Isolation Levels

There are several ways to isolate an application program from other application programs:

Repeatable Read

Database values read or changed by the program cannot be changed by other programs until the program reaches a point when it can COMMIT a *Logical Unit of Work* (or *Unit of Recovery*) The data can thus be read repeatedly without changing its value, until changes are committed.

When repeatable read is used, the users are protected from the scenario shown earlier, unless User A COMMITs the first change prior to the User B SELECT, then continues to UPDATE the same data without doing an intervening SELECT.

In ODBC, this is known as **Serializable**.

Read Stability

Like Repeatable Read, except that changes committed by other users will be noticed if a read is repeated.

In ODBC, this is known as **Repeatable Read**.

Cursor Stability

Database values read by the program are only protected while they are being used. As soon as the program moves from one row to the next, other programs may read or change the previous row. The database remains stable with respect to the position of the cursor. The use of cursor stability permits better concurrency (more users accessing the same tables, tablespaces, or dbspaces at the same time) but does not guarantee that the other data in a table remains consistent while you are working with individual rows.

In ODBC, this is known as **Read Committed**.

Uncommitted Read

Database values are not protected during read operations. This level is also sometimes known as *dirty read* because it allows one program to see values updated but not yet committed by another program.

During write operations, the behavior is the same as that of Cursor Stability.

In ODBC, this is known as **Read Uncommitted**.

Choose an isolation level with the ISOL request, or the APNAMES invocation option. See [SET ISOLATION LEVEL](#) for more information.

Fully-Qualified Update

Another technique to ensure that you are not interfering with someone else's update is the following:

1. Access the record.
2. COMMIT (this releases all locks).
3. Perform the analysis.
4. UPDATE, using a WHERE clause that specifies the *before* contents of all fields that you expect to update, as well as all the fields that uniquely identify a record.

For example:

1. `SELECT empno, position-code, salary`
`FROM employee-table`
`WHERE dept=12`
2. `COMMIT`
3. Perform the analysis, which determines that employee 12345 should have a change in position code to 38 and a raise to 2200.
4. `UPDATE employee-table`
`SET position-code=38, salary=2200`
`WHERE dept=12 AND empno=12345`
`AND position-code=37 AND salary=2000`

If the record is not found, the record was changed while you were doing the analysis. Retrieve it again, and try the update again.

If the update is successful, no fields of interest were changed and your update is safe.

The use of fully-qualified updates in APL2 requires fully cooperating programs and users. AP 127 runs as a single application program with PUBLIC authorization.

Table Locking

If you want to prevent other users from changing any row in a table while your APL function accesses the table's rows, your program can issue a LOCK statement. For example, suppose your program processes the data in several tablespaces but only one is sensitive to concurrent update problems. You can use isolation level CS during most operations, and then:

1. Issue the SQL statement `LOCK TABLE tablename` in SHARE mode.
2. Select the records of interest.
3. Perform the analysis.
4. Issue the update statements.
5. `COMMIT`.

The key ingredients here are the LOCK TABLE and COMMIT statements. The LOCK TABLE prevents any other user from updating the table until you issue the COMMIT statement. Until you issue the UPDATE statement, you have **SHARED** access: Authorized users can access the table but not update it. Between the time you issue the UPDATE statement and the time you issue the COMMIT, you have **EXCLUSIVE** access: Others are prevented from accessing any of the data in that table and any other tables in that tablespace. The COMMIT releases all locks.

If you are only accessing one table or tablespace, concurrency is better with repeatable read than with LOCK TABLE because repeatable read locks only the records retrieved, while LOCK TABLE locks the entire tablespace.

Note: In practice, all locking under TSO takes place at the page or tablespace level.

Under CMS, tables can be locked at the row, page, or dbspace level, using the LOCK command or the LOCK options of ALTER and ACQUIRE. If your dbspace is PRIVATE, locking takes place at the dbspace level.

Deadlock Handling

When two users attempt to update data in the same page, tablespace, or dbspace, a deadlock can occur. As a result, one of the users is not allowed to complete the update. In such a situation, a ROLLBACK is issued and an error code returned. This condition is considered a TRANSACTION BACKOUT. (Transaction backout can also occur for other reasons). All your updates since your last commit are lost. APL2 programs must be prepared to handle such a situation.

The program designer must carefully define the logical units of work or units of recovery. Small units of recovery promote concurrent access of data. When the data you have changed is consistent, it is time to issue a COMMIT.

Interrupting SQL Processing

On Workstation systems:

Interrupt interpreter processing. In the session manager you can use the interrupt option under the **Signals** pull-down. Otherwise, use the appropriate key combination for your operating system.

The interpreter interrupt is handled by APL2 in the normal manner. The database continues processing your request because the communication with the database is handled asynchronously.

To stop SQL processing: Retract the shared variable. As soon as control is returned to the auxiliary processor, the retract will be processed and a ROLLBACK will be issued for the current unit of work.

To continue SQL processing: Resume execution with →□LC.

On TSO systems:

Press the PA1 key twice.

The interpreter interrupt is handled by APL2 in the normal manner. The database continues processing your request because the communication with the database is handled asynchronously.

To stop SQL processing: Retract the shared variable. The subtask that communicates with the database is detached and the processing, if any, is stopped. A ROLLBACK will be issued for the current unit of work.

To continue SQL processing: Resume execution with →□LC.

On CMS systems:

Press the PA2 key.

If this action takes you into CMS, it means AP 127 was waiting for SQL when you pressed the PA2 key. While you are in CMS, you have the choice of one of two actions, depending on whether or not you want to stop SQL processing:

- To stop SQL processing:
Issue the SQLHX command. This results in:
 - A return code from SQL of -914 (user cancel)
 - A ROLLBACK issued for your current unit of work
 - Return control to your APL2 program
- To continue SQL processing:
Press the ENTER key. You will be returned to APL2 and SQL processing will resume.

If pressing PA2 does not take you into CMS, it means AP 127 was not waiting for SQL. The interrupt is handled by APL2 in the normal manner. See *APL2 Programming: System Services Reference* for more information about handling session interrupts under CMS.

Handling the LOB Datatypes

In recent versions of the workstation and TSO DB2 products, three new datatypes were introduced to allow for storage of large data objects. As a group, these datatypes are called the LOB datatypes. Individually, they are CLOB (character large object), BLOB (binary large object) and DBCLOB (double-byte character large object).

The CLOB and DBCLOB datatypes are like the VARCHAR and VARGRAPHIC types, respectively, except that their length can be longer. The BLOB datatype is similar to VARCHAR when declared with the FOR BIT DATA option. When data is sent by DB2 from one machine to another, no translation of BLOB data is performed, even if the machines are running with different character sets.

For more information on the SQL datatypes, see [SQL and APL2 Data Types](#).

Fetching LOB Data from DB2

When fetching data, CLOB and BLOB data is treated like VARCHAR data, and DBCLOB data like VARGRAPHIC data. The APL2 programmer does not need to do anything special for these datatypes in the SQL statements issued to fetch the data. Because the data in these columns can be quite large, however, some consideration may need to be given to storage requirements:

1. The APL2 workspace must be large enough to contain the fetched data.
2. APL2 shared memory must be large enough to pass the data through the shared variable.
3. The number of rows and columns being fetched on each call may need to be reduced to accomodate the increased storage requirements for the columns containing LOB data.

For information on workspace and shared memory allocation, see the *APL2 User's Guide* for your APL2 system. For information on controlling the number of rows fetched on each call, see [SETOPT](#).

Passing LOB Data to DB2

Internally, APL2 has two types of character data - regular character data and extended character data. If the APL2 programmer is using value lists to pass data, and regular character data is received from APL2, one of the SQL datatypes VARCHAR, LONG VARCHAR or CLOB is used, depending on the length of the data. When extended character data is received from APL2, VARGRAHIC, LONG VARGRAPHIC or DBCLOB is used, depending on the length of the data.

If the type of the target column is CLOB or DBCLOB, these data-passing conventions are adequate. DB2's rules of automatic data conversion allow conversion from any character type to CLOB, and from any graphic type to DBCLOB. If the type of the target column is BLOB, however, these conventions do not work. DB2's rules do not allow automatic conversion of any type to BLOB.

Since the auxiliary processor does not have knowledge of the type of the target columns when passing data, it does not have any way to determine that the BLOB datatype should be used. A special syntax has been defined for the APL2 programmer to indicate that the data is intended for a BLOB column. For example, the SQL statement

```
INSERT INTO TABLE VALUES (:1, :2, :3, :4)
```

is a simple insert statement with four target columns. If the third column is a BLOB column, the statement must be written as:

```
INSERT INTO TABLE VALUES (:1, :2, :B3, :4)
```

Either an upper-case or lower-case B may be used. It must immediately follow the colon and immediately precede the index number, with no intervening spaces.

Note:

Before the new syntax for value lists was defined in AP 127, a technique was documented for passing BLOB data that involved using the SQL CAST and BLOB built-in functions:

```
INSERT INTO TABLE VALUES (:1, :2, BLOB(CAST(:3 AS CLOB(n))), :4)
```

This technique may still be used, but it may not preserve the original data in all cases. For example, if a mainframe database is defined as using the ASCII character set, and the INSERT is done from the mainframe, the ASCII translation will be applied to the data before the BLOB function is applied.

Support for DBCS Characters

DBCS characters are supported by AP 127 and AP 227 in the following way:

1. You can pass extended character data in the value-list on an OPEN or CALL request. If the data contains extended characters, it is converted to VARGRAPHIC data before passing it to SQL. If it contains no extended characters, it is converted to SQL character data.
2. GRAPHIC, VARGRAPHIC and LONG VARGRAPHIC data in SQL tables can be retrieved into APL, and is returned as extended character data.
3. (Mainframe Systems Only) You can use extended characters in SQL statements. The statements are converted to SO/SI format and passed to SQL.
4. (Mainframe Systems Only) Retrieval of AP 127 messages translated to DBCS are supported. The translations of the messages, however, are not included with APL2.

APL2/SQL Interface Messages

This section describes:

- [Auxiliary Processor Messages](#)
- [SQL Workspace Messages](#)

Messages from SQL are explained in the *Messages and Codes* manual for your database system. These messages are identified by a return code in the form

1 0 0 2 *code* for ERROR conditions
0 1 0 2 *code* for WARNING conditions
1 1 0 2 *code* for TRANSACTION BACKOUT

The 2 (fourth item) indicates that the fifth item contains an SQL return code.

Auxiliary Processor Messages

The following messages are produced by AP 127 and AP 227. They are identified by a return code in the form:

1 0 0 1 *msgn* for ERROR conditions

0 1 0 1 *msgn* for WARNING conditions

The 1 (fourth item) indicates that the fifth item contains the number of a message from the auxiliary processor.

The message identifiers used here are composed of the characters AP2X127 followed by the 3-digit number in *msgn*. On mainframe systems, message text retrieved or displayed is automatically prefixed with the message identifier when invocation option `DEBUG(1)` is in effect.

On mainframe systems, the errors associated with messages AP2X127103 through AP2X127104, AP2X127106, and AP2X127148 may cause AP 127 to terminate processing. In this case, the text of the message is displayed on the terminal before processing terminates or when AP 127 restarts. The associated abnormal termination (abend) codes are 2103 through 2104, 2106, and 2148. The abend codes are printed if processing terminates.

Italicized expressions in the message text represent descriptive tokens, which are inserted into the text.

All message numbers are between 103 and 180.

AP2X127103 INSUFFICIENT STORAGE HAS BEEN ALLOCATED BY THIS PROCESSOR

Explanation: The amount of internal storage allocated by the auxiliary processor for subroutine calls is not large enough. This is an internal logic problem.

System Action: The auxiliary processor may abend.

User Response: Contact your system administrator.

AP2X127104 APL2 PROCESSOR ERROR IN DATA FORMAT

Explanation: The data passed to the auxiliary processor by the shared variable processor (SVP) is not in standard APL2 format. This is an internal APL2 logic problem.

System Action: The auxiliary processor may abend.

User Response: Contact your system administrator.

AP2X127105 SQL DATA TYPE *number* (COLUMN *name*) NOT SUPPORTED BY AP 127

Explanation: The data being fetched contains a column with a data type not supported by the auxiliary processor.

System Action: The request is not processed.

User Response: Rework the query so that it does not include the named column.

AP2X127106 AP 127 PROCESSOR ERROR - INVALID INTERNAL CODE

Explanation: The auxiliary processor has been called with an operation code it does not recognize. This is an internal logic problem.

System Action: The auxiliary processor mayabend.

User Response: Contact your system administrator.

AP2X127109 THE RESULT TABLE IS TOO LARGE FOR THE SHARED VARIABLE

Explanation: The result table retrieved by a FETCH request is too large to fit into the space allocated for the shared variable.

System Action: The result table is not placed in the shared variable.

User Response: Reissue the FETCH request (or issue the SETOPT request), specifying fewer rows in the options list, or increase the available shared memory. On mainframe systems, this is controlled by the SHRSIZE invocation option. On workstation systems, it is controlled by the options in the file `apl2svp.prm`.

Note: Since the data has already been FETCHed, you need to reposition the cursor and start the FETCHing over.

AP2X127110 THE ERROR CODE IS NOT A 5-ITEM NUMERIC VECTOR

Explanation: The return code item passed through a 'MSG' operation or MESSAGE function is not a 5-item numeric vector.

System Action: The command is not processed.

User Response: Respecify the return code vector.

AP2X127111 ATTEMPT TO SHARE MORE THAN THE ALLOWED LIMIT OF *number* VARIABLES WITH AP 127

Explanation: You have attempted to share more variables with the auxiliary processor than it is prepared to accept at one time. The current limit for mainframe systems is 10 shared variables.

System Action: The additional variable is not matched.

User Response: Specify a variable already shared, or retract one or more variables before sharing again.

AP2X127112 THE FIRST ITEM OF THE SHARED VARIABLE IS NOT A RECOGNIZED OPERATION

Explanation: The operation code passed to the auxiliary processor is not one of the recognized character-string operation codes.

System Action: The command is not processed.

User Response: Check the spelling of the operation code, and respecify the variable.

AP2X127113 THE *operation* OPERATION EXPECTS *num1* ARGUMENTS BUT RECEIVED *num2*

Explanation: The number of arguments for the operation is too great or too small.

System Action: The command is not processed.

User Response: Respecify the variable with the correct number of arguments for the operation.

AP2X127114 SQL STATEMENT NOT CHARACTER VECTOR OR MATRIX

Explanation: An SQL statement was passed through the PREP or EXEC operation, but it was not a simple character vector or matrix.

System Action: The command is not processed.

User Response: Resubmit the request with the statement in the correct form.

AP2X127115 ATTEMPT TO USE OTHER THAN ACTIVE SHARED VARIABLE *name*

Explanation: An attempt has been made to specify a second shared variable before the first has been retracted. On some systems, multiple variables may be shared, but only one can be *active* (specified) at a time. After a variable has become active, it remains so until it is retracted.

System Action: The command is not processed.

User Response: Retract the first variable before specifying the second.

AP2X127119 *number* IS NOT A RECOGNIZED ROWS VALUE

Explanation: The number entered for rows in an options list is not a positive integer.

System Action: The command is not processed.

User Response: Respecify the number of rows.

AP2X127120 ATTEMPT TO PREPARE MORE THAN THE ALLOWED LIMIT OF *number* STATEMENTS

Explanation: No more than *number* SQL statements can be processed using the DECLARE, PREP and ODBCOPEN requests before one or more of them must be purged. The current limit is 40 prepared statements.

System Action: The command is not processed.

User Response: Issue a PURGE request for one or more of the existing statements, or reuse a statement name in a PREP request.

AP2X127121 INSUFFICIENT STORAGE TO *token1 token2*

Explanation: The auxiliary processor cannot obtain enough space to honor the named request. *token1* is PREPARE, DECLARE, DESCRIBE, ODBCOPEN or EXECUTE. If *token1* is PREPARE, DECLARE, DESCRIBE or ODBCOPEN, *token2* is the statement name; if *token1* is EXECUTE, *token2* is IMMEDIATE.

System Action: The command is not processed.

User Response: Issue a PURGE request for one or more SQL statements to make space available. On mainframe systems, you may be able to increase the amount of free storage by adjusting the WSSIZE and/or FREESIZE invocation options, or by allocating a larger machine or region to your user id.

AP2X127122 INDEX VALUE FOUND IN STATEMENT POSITION *number* IS NOT OF A RECOGNIZABLE TYPE

Explanation: An index value in an SQL statement is not a positive integer.

System Action: The command is not processed.

User Response: Change the index value to a positive integer.

AP2X127125 INDEX MARKER FOUND IN POSITION *number* OF AN EXEC STATEMENT

Explanation: The SQL statement passed with an EXEC request contains a colon marking an item as a vector index. EXEC requests cannot contain vector indexes.

System Action: The command is not processed.

User Response: Correct the SQL statement, and reissue the EXEC request.

AP2X127126 NO ACTIVE STATEMENTS EXIST

Explanation: A GET NAMES request could not be completed because no SQL statements have been processed using the DECLARE, PREP or ODBCOPEN requests.

System Action: A null prototype of the name list is returned.

User Response: Issue a DECLARE, PREP or ODBCOPEN request before issuing a NAMES request.

AP2X127127 UNKNOWN OPTION VALUE

Explanation: The options list contains a character value other than 'MATRIX', 'VECTOR', 'LENGTH', or 'NOLENGTH'.

System Action: The command is not processed.

User Response: Correct the options-list, and reissue the request.

AP2X127131 *number* IS AN UNRECOGNIZED ERROR CLASS

Explanation: The fourth item of the numeric vector argument to the 'MSG' operation is not 1 or 2.

System Action: The command is not processed.

User Response: Correct the fourth item of the argument, and reissue the request.

AP2X127132 *number* IS AN UNDEFINED MESSAGE NUMBER FOR AP 127

Explanation: Although the fourth item is 1, the fifth item of the numeric vector argument to the 'MSG' operation or MESSAGE function is not a valid auxiliary processor message number.

System Action: The command is not processed.

User Response: Correct the fifth item of the argument, and reissue the request, or check the fourth item to be sure that it correctly identifies the source of the error - 1 for the auxiliary processor, 2 for SQL, or 3 for a workspace function (permitted only with the MESSAGE function).

AP2X127133 INSUFFICIENT STORAGE FOR VALUE-LIST OF *name*

Explanation: The auxiliary processor cannot allocate enough space to format the value-list passed on a CALL or OPEN request.

System Action: The command is not processed.

User Response: Decrease the size of the value-list. On mainframe systems, you may be able to increase the amount of free storage by adjusting the WSSIZE and/or FREESIZE invocation options, or by allocating a larger machine or region to your user id.

AP2X127134 THE STATEMENT *name* EXPECTS AT LEAST *num1* ITEMS IN ITS VALUE-LIST AND RECEIVED *num2*

Explanation: The value-list passed with a CALL or an OPEN request has fewer items than the highest index value passed with the statement.

System Action: The command is not processed.

User Response: Either increase the number of items in the value-list, or change the vector index to reflect the actual number of items passed.

AP2X127135 *name* IS A SELECT STATEMENT - *operation* OPERATION CANNOT BE COMPLETED

Explanation: A CALL or PUT request has been issued for a cursor SELECT statement.

System Action: The command is not processed.

User Response: Correct the name in the request or issue an OPEN request for the statement.

AP2X127136 *name* IS NOT A SELECT STATEMENT - *operation* OPERATION CANNOT BE COMPLETED

Explanation: A DESCRIBE or OPEN request has been issued for a statement other than an SQL SELECT statement or a stored procedure call.

System Action: The command is not processed.

User Response: Correct the name in the OPEN request or issue a CALL request for the statement.

AP2X127137 *name* IS NOT PREPARED - *operation* OPERATION CANNOT BE COMPLETED

Explanation: A CALL, OPEN, or DESCRIBE request has been made for a statement that has not been processed by a PREP or ODBCOPEN request, or an OPEN request has been made for a stored procedure call that has not been processed by a CALL request.

System Action: The command is not processed.

User Response: Issue the correct sequence of commands for the statement named.

AP2X127139 *name* IS NOT OPEN - *operation* OPERATION CANNOT BE COMPLETED

Explanation: A FETCH or CLOSE request has been issued for a statement that has not been processed using an OPEN or ODBCOPEN request.

System Action: The command is not processed.

User Response: Issue the OPEN request before issuing the FETCH or CLOSE.

AP2X127141 THE VALUE PASSED IN POSITION *number* IS OF A TYPE NOT RECOGNIZABLE BY SQL

Explanation: The item identified, for example, a complex number, cannot be processed by SQL.

System Action: The command is not processed.

User Response: Correct the value-list, and reissue the request.

AP2X127142 THE VALUE-LIST PASSED IS IN AN UNRECOGNIZED FORMAT

Explanation: The value-list argument in a CALL or OPEN request is not a numeric vector, a character vector, a numeric scalar, or a mixed vector of character vectors and numeric scalars. The depth must be 2 or less.

System Action: The command is not processed.

User Response: Correct the value-list, and reissue the request.

AP2X127143 STATEMENT NAME OF LENGTH *length* IS TOO LONG

Explanation: The name specified for a statement must fit in 18 bytes of storage after removing leading and trailing blanks.

System Action: The command is not processed.

User Response: Shorten the name, and reissue the PREP request.

AP2X127144 STATEMENT NAME NOT VALID

Explanation: The name specified for an SQL statement on a PREP request is not a character vector with length 2 through 18 and a nonunderbarred alphabetic character as its first item.

System Action: The command is not processed.

User Response: Correct the name, and reissue the PREP request.

AP2X127145 THE STATEMENT *name* DOES NOT EXIST

Explanation: The SQL statement named has not been declared.

System Action: The command is not processed.

User Response: Either correct the name, or issue a DECLARE, PREP or ODBCOPEN request for the name.

AP2X127146 WARNING - VALUE LIST LONGER THAN NECESSARY

Explanation: A value-list passed on an OPEN or CALL request has more items than the highest vector index in the statement. This is a warning only. It does not necessarily indicate an error.

System Action: The command is processed normally.

User Response: Verify that the value-list is correctly formatted.

AP2X127147 INSUFFICIENT STORAGE TO INITIALIZE DATABASE

Explanation: The auxiliary processor could not obtain enough storage to contain the database parameter area.

System Action: The database connection is not completed.

User Response: On mainframe systems, you may be able to increase the amount of free storage by adjusting the WSSIZE and/or FREESIZE invocation options, or by allocating a larger machine or region to your user id.

AP2X127148 INSUFFICIENT STORAGE TO ALLOCATE DATA BUFFER OF LENGTH *number*

Explanation: The auxiliary processor could not obtain enough storage for the shared variable data buffer size required. This could be the result of either the size of the user request or the size of the result table.

System Action: The request is not processed.

User Response: Decrease the size of the object being passed or retrieved, or increase the amount of storage available (see message AP2X127147).

AP2X127149 SQL STATEMENT OF LENGTH *number* IS TOO LONG

Explanation: The SQL statement passed is longer than the maximum allowed by the database.

System Action: The request is not processed.

User Response: Shorten the SQL statement.

AP2X127150 VALUE PASSED IN POSITION *num1* OF LENGTH *num2* IS TOO LONG

Explanation: One of the items in the value-list passed on an OPEN or CALL request is larger than the maximum allowed by the database.

System Action: The request is not processed.

User Response: Check the value and correct it.

AP2X127152 THE AP 127 - DB2 SUBTASK HAS ABENDED WITH CODE *number*

Explanation: The subtask attached by the auxiliary processor to communicate with the database system has abended.

System Action: All prepared statements are purged. The auxiliary processor may still be used. An ATTACH

command is reissued if needed.

User Response: See your system programmer.

AP2X127153 ERROR DURING DB2 CALL - RETURN CODE: *num1* REASON CODE: *num2*

Explanation: An error was encountered during a database call, which could not be translated into an SQLCODE. The DB2 return and reason codes are documented in *IBM DATABASE 2 Messages and Codes*.

System Action: The database connection is not completed.

User Response: If the DB2 documentation does not contain sufficient information to determine the source of the error, see your database administrator.

AP2X127154 operation OF AP 127 SUBTASK FAILED - RETURN CODE: *number*

Explanation: The subtask that calls DB2 could not be attached because of the failure of the indicated TSO operation.

System Action: The database connection is not completed.

User Response: Contact your system administrator.

AP2X127155 DB2 OPERATOR STOPPING SUBSYSTEM - LINK TERMINATED

Explanation: The DB2 operator is taking your subsystem down. You cannot continue processing.

System Action: The database connection is terminated.

User Response: See your database administrator.

AP2X127156 DB2 STOPPING ABNORMALLY - LINK TERMINATED

Explanation: DB2 is terminating abnormally. You cannot continue processing.

System Action: The database connection is terminated.

User Response: See your database administrator.

AP2X127157 DB2 OPERATOR STOPPING SUBSYSTEM - PLEASE END YOUR SESSION

Explanation: The DB2 operator is taking your subsystem down but is delaying to allow you to end your application normally.

System Action: The database connection will be terminated shortly.

User Response: Finish your unit of work, and issue a COMMIT as quickly as possible.

AP2X127158 AP 127 IS RESTARTED DUE TO ABEND - CODE: *number* TYPE: *character*

Explanation: An ABEND has been issued for the auxiliary processor. *number* is the numeric code. *character* is the type of error:

P for programmer

S for system

U for user

System Action: The auxiliary processor is restarted. All statements are purged and the connection with the

database is broken.

User Response: Contact your system administrator.

AP2X127159 DATABASE INTERFACE LINKAGE NOT INSTALLED WITH APL2

Explanation: The auxiliary processor cannot be accessed because prerequisite parts of the database system have not been installed with APL2.

System Action: The command is not processed.

User Response: On CMS, the text deck ARIRVSTC must be link-edited into the APL2 loadlib. Contact your system administrator.

AP2X127161 *command* IS NOT VALID IN THIS OPERATING SYSTEM

Explanation: A syntactically-correct command has been issued, but the operating system does not allow it. For example, the PUT command is not allowed except on CMS, and SSID command is allowed only in TSO.

System Action: The command is not processed.

User Response: Do not issue the command in that environment.

AP2X127162 THE *argument* ARGUMENT TO CONNECT MUST BE A CHARACTER VECTOR OF LENGTH 1 TO *number*

Explanation: An argument has been passed to CONNECT that is either not character or not of correct length.

System Action: The command is not processed.

User Response: Correct the argument and reissue the request.

AP2X127163 UNEXPECTED RETURN CODE *number* FROM *routine*

Explanation: The auxiliary processor received an unexpected return code from a database routine. For example, on TSO, in processing a 'MSG' operation, an unexpected return code was received from the DB2 routine DSNTIAR. There is probably an internal auxiliary processor error.

System Action: This message is issued as a warning. The 'MSG' operation is completed, and the SQLCA contents are still returned. The message text, however, is missing.

User Response: Contact your system programmer.

AP2X127164 ERROR LOADING DB2 PROGRAM *name*

Explanation: The auxiliary processor was unable to load the named program.

System Action: The connection to the database is not established.

User Response: Check that the appropriate database libraries have been accessed by your user id, either in LPALIB, your LOGON PROC, or the CLIST used to invoke APL2.

AP2X127165 INVALID PARAMETER TO *operation* OPERATION

Explanation: The parameter passed to the ISOL operation is not CS, RR, RS or UR; the parameter passed to the DESCRIBE operation is not NAMES, LABELS, BOTH or ANY; the parameter passed to the DECLARE operation is not HOLD or NOHOLD; the parameter passed to the ODBC operation is not DATASOURCES or DRIVERS; the parameter passed to the ODBCOPEN operation is not GETTYPEINFO or TABLES; the parameter passed to the XMLSIZE operation is not a single integer; the parameter passed to the COMMIT or ROLLBACK operation is not RELEASE.

System Action: The command is not processed.

User Response: Correct the call and reissue.

AP2X127166 INVALID EXTENDED CHARACTERS

Explanation: The extended characters passed have been determined to be invalid. This can occur if the character set ID in the data does not match the character set ID set for the APL2 session (through the DBCS invocation option), or if the data contains characters not within the defined range for DBCS characters.

System Action: The request is not processed.

User Response: Verify the data and resubmit the request.

AP2X127167 CANNOT SWITCH SUBSYSTEMS WHEN CONNECTED

Explanation: The SSID command has been issued when the database connection is already active.

System Action: The command is not processed.

User Response: Retract the shared variable or issue a ROLLBACK or COMMIT with the RELEASE option before issuing the SSID command.

AP2X127180 NO VALUE-LIST PASSED FOR PUT *name*

Explanation: A PUT command was issued without a value-list. A value-list is required for this command.

System Action: The command is not processed.

User Response: Reissue the command with a value-list.

SQL Workspace Messages

The following messages are issued by the SQL workspace functions.

Messages 1 and 2 are issued as APL2 errors using □ES. For information of how to diagnose these messages, see *APL2 Programming: Language Reference*. The remainder of the workspace error messages are identified by a return code in the form:

1 0 0 3 *msgn*

The 3 as the fourth item indicates that the fifth item contains the number of a message from the SQL workspace. The message text can be returned to the workspace only through the MESSAGE function. If MORE is entered after a call to MESSAGE, a fully-qualified error code in the form

AP2WSQL10

is printed, where:

AP2W Identifies an APL2 distributed workspace message.

SQL Is the name of the distributed workspace.

10 Is the error number.

The following message:

ERROR AP2WSQL *msgn* HAS OCCURRED

is returned if the workspace error code is unknown.

Note: The workspace functions do not attempt to analyze all argument errors; when possible, further analysis is left to the auxiliary processor.

1 ERROR DURING OFFER

Explanation: An offer to share DAT returned a 0.

System Action: The request is not processed.

User Response: Check the name you have used for validity and conflict with other names already in the workspace.

2 UNABLE TO OFFER

Explanation: An offer to share DAT was not coupled after successive tries.

System Action: The request is not processed.

User Response: Verify that AP 127 or AP 227 is correctly installed with your APL2 system.

3 'DAT' IS NOT SHARED WITH AP 127

Explanation: A function was called that requires a prior coupling of the variable DAT but DAT is not coupled.

System Action: The request is not processed.

User Response: Check the sequence of operations you are performing for validity. A call to the OFFER function is included in other functions where it is appropriate to begin a sequence of calls with those functions.

4 ITEM IS EMPTY, ALL ZERO, OR ALL BLANK

Explanation: An argument to a workspace function is empty, all zeros, or all blank, but the function requires an argument with values.

System Action: The request is not processed.

User Response: Correct the argument and reissue the request.

5 ARGUMENT MUST BE A VECTOR

Explanation: The workspace function requires a vector argument.

System Action: The request is not processed.

User Response: Correct the argument and reissue the request.

6 RIGHT ARGUMENT LENGTH ERROR

Explanation: The right argument to a workspace function (SHOW, SQL, QUE) contains the wrong number of items.

System Action: The request is not processed.

User Response: Correct the argument and reissue the request.

7 STATEMENT ITEM MUST BE SIMPLE

Explanation: The SQL statement passed to the SQL function contains an item that is not a simple scalar, vector, or matrix.

System Action: The request is not processed.

User Response: Correct the item and reissue the request.

8 ARGUMENT MUST BE SCALAR OR VECTOR

Explanation: The right argument of an access function (OPEN or CALL, for example) must be a simple vector, an enclosed simple vector, or a depth 2 vector.

System Action: The request is not processed.

User Response: Correct the argument and reissue the request.

9 ARGUMENT MUST NOT BE EMPTY

Explanation: The workspace function requires a nonempty argument.

System Action: The request is not processed.

User Response: Correct the argument and reissue the request.

10 ARGUMENT MUST NOT BE SIMPLE

Explanation: The right argument of the PREP function must not be simple.

System Action: The request is not processed.

User Response: Correct the argument and reissue the request.

11 FIRST ITEM IS NOT A CHARACTER STRING

Explanation: The SQL function requires a character string as the first item of its argument.

System Action: The request is not processed.

User Response: Specify a character string as the first item and reissue the request.

12 CHARACTER STRING LENGTH IS ONE

Explanation: The workspace function SQL requires an SQL statement; it must contain more than one character.

System Action: The request is not processed.

User Response: Respecify the character string and reissue the request.

13 WRONG NUMBER OF ARGUMENTS FOR THIS STATEMENT

Explanation: The SQL function does not allow a value-list with an EXEC immediate statement.

System Action: The request is not processed.

User Response: Correct the arguments and reissue the request.

14 MESSAGE CODE MUST BE A SIMPLE LENGTH 5 VECTOR

Explanation: The MESSAGE function requires a simple vector of length 5 as an argument.

System Action: The request is not processed.

User Response: Correct the argument and reissue the request.

15 STACK MUST BE A MATRIX

Explanation: The stack passed by RESUME to QUE must be a one-column matrix.

System Action: The request is not processed.

User Response: Correct the argument and reissue the request.

16 CHART CALL HAS FAILED. RETURN CODE IS *code*

Explanation: A call to the Interactive Chart Utility has failed. *code* is the return code from GDDM.

System Action: The request is not processed.

User Response: The GDDM return codes are documented in *GDDM: Messages*.

Quick Reference Tables

This appendix contains tables that summarize reference information useful when working with the APL2/SQL interface:

- [Auxiliary Processor Operation Codes](#) summarizes the auxiliary processor operation codes and their syntax, and indicates whether an equivalent SQL workspace function exists.
- [SQL Workspace Functions](#) summarizes the SQL workspace functions and operator, their syntax, and indicates the equivalent auxiliary processor operations.
- [Return Code Vector](#) summarizes the return codes from auxiliary processor and SQL workspace operations.
- [SQL Statement Types and APL2 Operations](#) summarizes SQL statement types, SQL statements, and the auxiliary processor operations that can be used to process them.
- [SQL and APL2 Data Types](#) summarizes the SQL datatypes and their relationship to APL2 datatypes.

Auxiliary Processor Operation Codes

Operation Code and Syntax	Workspace Function
'CALL' <i>name</i> [<i>values</i>]	CALL
'CLOSE' <i>name</i>	CLOSE
'COMMIT' ['RELEASE']	COMMIT
'CONNECT' <i>database-identifier</i>	CONNECT
'DECLARE' <i>name</i> ['HOLD' 'NOHOLD']	DECLARE
'DESCRIBE' <i>name</i> [<i>type</i>]	DESC
'EXEC' <i>stmt</i>	EXEC
'FETCH' <i>name</i> [<i>options</i> ..]	FETCH
'GETOPT'	GETOPT
'ISOL' [<i>setting</i>]	ISOL
'MSG' <i>rcode</i>	MESSAGE
'NAMES'	NAMES
'ODBC' <i>type</i>	ODBC
'ODBCOPEN' <i>name type</i>	ODBCOPEN
'OPEN' <i>name</i> [<i>values</i>]	OPEN
'PREP' <i>name stmt</i>	PREP
'PURGE' <i>name</i>	PURGE
'PUT' <i>name values</i>	PUT
'ROLLBACK' ['RELEASE']	ROLLBACK
'SETOPT' <i>options</i> ..	SETOPT
'SQLCA'	SQLCA
'SQLSTATE'	SQLSTATE
'SSID' [<i>subsystem</i>]	SSID
'STATE' <i>name</i>	STATE
'STMT' <i>name</i>	STMT
'TRACE' [(<i>module level</i>)..]	TRACE
'XMLSIZE' [<i>size</i>]	XMLSIZE

SQL Workspace Functions

Function Name and Syntax	Auxiliary Processor Operation
CALL <i>name</i> [<i>values</i>]	'CALL '
CHART <i>data</i>	
CLOSE <i>name</i>	'CLOSE '
COMMIT	'COMMIT '
CONNECT <i>database-identifier</i>	'CONNECT '
<i>format</i> DATE <i>ts</i>	
DECLARE <i>name</i> ['HOLD ' 'NOHOLD ']	'DECLARE '
DESC <i>name</i> [<i>type</i>]	'DESCRIBE '
EVAL <i>data</i>	
EVALSIM <i>data</i>	
EXEC <i>stmt</i>	'EXEC '
FETCH <i>name</i> [<i>options..</i>]	'FETCH '
GETOPT	'GETOPT '
IN	
ISOL <i>setting</i>	'ISOL '
MESSAGE <i>rcode</i>	'MSG '
NAMES	'NAMES '
ODBC <i>type</i>	'ODBC '
ODBCOPEN <i>name type</i>	'ODBCOPEN '
OFFER	
OPEN <i>name</i> [<i>values</i>]	'OPEN '
PREP <i>name stmt</i>	'PREP '
PROCEDURE <i>stmt</i> [<i>values</i>]	
PURGE <i>name</i>	'PURGE '
PUT <i>name values</i>	'PUT '
QUE <i>stack</i>	
QUERY <i>stmt</i> [<i>values</i>]	
RESUME <i>stack</i>	
ROLLBACK	'ROLLBACK '
SETOPT <i>options..</i>	'SETOPT '
SHOW <i>result</i>	
SQL <i>stmt</i> [<i>values</i>]	
SQLCA	'SQLCA '
SQLHELP <i>keyword</i>	

Function Name and Syntax	Auxiliary Processor Operation
SQLSTATE	'SQLSTATE '
SSID <i>subsystem</i>	'SSID '
STATE <i>name</i>	'STATE '
STMT <i>name</i>	'STMT '
<i>format</i> TIME <i>ts</i>	
TIMESTAMP <i>ts</i>	
TRACE (<i>module level</i>)..	'TRACE '
(<i>f</i> UNTIL) <i>stack</i>	
XMLSIZE <i>size</i>	'XMLSIZE '

Return Code Vector

Return Code Vector	Meaning
0 0 0 0 0	Normal return. All operations completed. Result table retrieved by a FETCH request is complete.
0 0 1 0 0	Normal return, but a result table may not have been completely retrieved.
1 0 0 1 <i>msgn</i>	Error in auxiliary processor. <i>msgn</i> is the number of the auxiliary processor error message.
1 0 0 2 <i>msgn</i>	Error detected in the database system. <i>msgn</i> gives the SQL return code (SQLCODE).
1 0 0 3 <i>msgn</i>	Error detected in an SQL workspace function. <i>msgn</i> gives the message number.
0 1 0 <i>n msgn</i>	Warning message. For example, FETCH has no more rows to retrieve, a DELETE statement deletes nothing, or the value-list is longer than the highest vector index.
1 1 0 <i>n msgn</i>	Transaction backout. All changes made to tables since the last COMMIT or ROLLBACK have been discarded. Application must restore processing to point of last COMMIT or ROLLBACK. All locks are released and all cursors closed.

SQL Statement Types and APL2 Operations

SQL Statement Type	SQL Statement	Processing Method
Query	SELECT	<p>The following sequence only:</p> <p>DECLARE (optional), PREP, OPEN, FETCH, CLOSE</p> <p>Values for SQL clauses are embedded or passed as a vector on the OPEN request.</p>
Data Manipulation	DELETE INSERT UPDATE	<p>Either:</p> <p>EXEC (values must be embedded in SQL statement)</p> <p>or:</p> <p>PREP, CALL (values are either embedded or passed as APL2 data on the CALL request)</p> <p>In CMS only, the PREP, OPEN, PUT, CLOSE sequence can also be used for INSERT.</p>
Data Definition	CREATE ALTER DROP ACQUIRE COMMENT ON LABEL ON	EXEC request
Authorization	GRANT REVOKE	EXEC request
Control	LOCK TABLE COMMIT ROLLBACK CONNECT	<p>EXEC request</p> <p>COMMIT, ROLLBACK, and CONNECT are submitted directly to the database management system. They are not only SQL statements, but also APL2/SQL interface requests.</p>
Procedure Call	CALL	<p>PREP, CALL (values must be passed as APL2 data on the CALL request)</p> <p>Then, for each result set built by the procedure:</p> <p>OPEN, FETCH, CLOSE</p>
Analysis	EXPLAIN	EXEC request

SQL Data Types

Note: Not all of these SQL datatypes are supported by all databases. Check your database documentation for information on which types it supports.

SQL Data Type	Definition	APL2 Data Type
BIT	Single-bit binary data.	Real, numeric integer
TINYINT	One-byte integer with a precision of 3 digits. Range is -128 to +127.	Real, numeric integer
SMALLINT	Two-byte integer with a precision of 5 digits. Range is -32768 to +32767.	Real, numeric integer
INTEGER	Four-byte integer with a precision of 10 digits. Range is -2147483648 to +2147483647	Real, numeric integer
BIGINT	Eight-byte integer with a precision of 19 digits. Range is -9223372036854775808 to +9223372036854775807.	Real, numeric
REAL	Single Precision (4-byte) floating-point number.	Real, numeric
FLOAT	Double-precision (8-byte) floating-point number.	Real, numeric
DOUBLE	Double-precision (8-byte) floating-point number. (Same as FLOAT)	Real, numeric
DECIMAL (<i>m,n</i>)	Packed or Zoned decimal, specified as <i>m</i> for overall column width (precision) and <i>n</i> for number of digits to the right of the decimal (scale).	Real, numeric
NUMERIC (<i>m,n</i>)	Signed, exact numeric, specified as <i>m</i> for overall column width (precision) and <i>n</i> for number of digits to the right of the decimal (scale). (Same as DECIMAL)	Real, numeric
	Not supported by SQL	Complex
CHAR (<i>n</i>)	Fixed-length character, where <i>n</i> is the number of characters with a limit of 254 characters.	Character
VARCHAR (<i>n</i>)	Variable-length character, where <i>n</i> is the maximum column width. Maximum width is system-dependant. For DB2, it is not greater than 4000 on workstations, 32767 on CMS and the system-defined page size on TSO.	Character
LONG VARCHAR	Variable-length character. Maximum width is system-dependant. For DB2, it is not greater than 32700 on workstations, 32767 on CMS and the system-defined page size on TSO.	Character
CLOB (<i>n</i>)	Character large object. Variable-length character, where <i>n</i> is the maximum column width. Maximum width is 2147483647.	Character

SQL Data Type	Definition	APL2 Data Type
BINARY (<i>n</i>)	Fixed-length binary character, where <i>n</i> is the number of bytes with a limit of 254 bytes.	Character
VARBINARY (<i>n</i>)	Variable-length binary character, where <i>n</i> is the maximum column width. Maximum width is data source-dependant.	Character
LONG VARBINARY	Variable-length binary character. Maximum width is data source-dependant.	Character
BLOB (<i>n</i>)	Binary large object. Variable-length binary character, where <i>n</i> is the maximum column width. Maximum width is 2147483647.	Character
GRAPHIC (<i>n</i>)	Fixed length double-byte characters; where <i>n</i> is the number of characters with a limit of 127 characters.	Extended character
VARGRAPHIC (<i>n</i>)	Variable length double-byte characters, with maximum column width specified as <i>n</i> . Maximum width is system-dependant. For DB2, it is not greater than 2000 on workstations, 16383 on CMS and one-half the system-defined page size on TSO.	Extended character
LONG VARGRAPHIC	Variable length double-byte characters. Maximum width is system-dependant. For DB2, it is not greater than 16350 on workstations, 16383 on CMS and one-half the system-defined page size on TSO.	Extended character
WIDE CHAR (<i>n</i>)	Fixed length double-byte Unicode characters; where <i>n</i> is the number of characters with a limit of 127 characters.	Extended character
WIDE VARCHAR (<i>n</i>)	Variable length double-byte Unicode characters, with maximum column width specified as <i>n</i> . Maximum width is system-dependant.	Extended character
WIDE LONG VARCHAR	Variable length double-byte Unicode characters. Maximum width is system-dependant.	Extended character
DBCLOB (<i>n</i>)	Double-byte character large object. Variable-length double-byte characters, with maximum column width specified as <i>n</i> . Maximum width is 1073741823.	Extended character
DATE	Date as defined by the database.	Character
TIME	Time as defined by the database.	Character
TIMESTAMP	Timestamp as defined by the database.	Character
XML	XML data. Notes on APL2 processing of the XML data type:	Character

SQL Data Type	Definition	APL2 Data Type
	<ol style="list-style-type: none"> 1. Since the XML column type does not have a maximum size associated with it, the amount of storage allocated by APL2 for fetching XML data is user-controlled, with the XMLSIZE command. If no size is specified by the user, the default size used is 65536. 2. XML data is fetched into APL2 using the BLOB binary datatype. If a value-list is used to pass APL2 character data to DB2 for an XML column, the application may want to force use of the BLOB type for input as well, to avoid any translation of the data. For more information on forcing use of the BLOB type, see Passing LOB Data to DB2. 	
ROWID	Row identifier (TSO only).	Character

More information on these SQL data types can be found in the *SQL Reference* for your database system.

Note: Since the data types of SQL and APL2 do not correspond exactly, SQL does some implicit data conversions when data is passed to and from the database. For example, APL2 may internally represent data that appears to be integer as floating (for example, 2 as 1.9999999999999999). APL2 also has no BIGINT, REAL, or DECIMAL types, so SQL converts these types to and from FLOAT. Some loss of precision may result from these conversions.

Glossary

access function

One of the SQL functions: EXEC, PREP, CALL, OPEN, FETCH, CLOSE.

access operations

Auxiliary processor operations that provide access to the database system, specifically, 'EXEC', 'PREP', 'CALL', 'OPEN', 'FETCH', 'CLOSE'.

access request

Either an SQL workspace access function or its corresponding auxiliary processor operation.

access sequence

Any sequence of access requests.

active variable

The first variable specified after one or more variables have been shared (coupled) with AP 127. On some systems, only one variable can be active at a time.

APL2/SQL interface

Two facilities - the Structured Query Language processor (AP 127 or AP 227) and the SQL workspace - that provide access to relational tables through the SQL language. See also *auxiliary processor* and *SQL workspace*.

argument

An array parameter that is passed to a function (to be distinguished from an operand of an operator).

array

An array is an ordered collection of items. The arrays of APL2 are finite rectangular arrays that contain arrays as items. See *APL2 Programming: Language Reference* for a further description of APL2 arrays.

authorization statement

An SQL GRANT or REVOKE statement that gives to or removes from a user certain database access privileges.

auxiliary processor, AP 127

The IBM-supplied processor that handles shared variable communications between the active APL2 workspace and the DB2 and SQL/DS database management systems.

auxiliary processor, AP 227

The IBM-supplied processor that handles shared variable communications between the active APL2 workspace and the ODBC (Open Database Connectivity) Driver Manager.

completion code

First three items of the return code vector; they indicate whether the request to the database management system completed successfully.

control statement

An SQL LOCK TABLE, CONNECT, COMMIT, or ROLLBACK statement that aids in managing a logical unit of work. LOCK TABLE is submitted through an EXEC request. CONNECT, COMMIT and ROLLBACK are not only SQL statements but also auxiliary processor operations.

cursor

Pointer to a row of a result table created by processing an SQL select statement through a PREP request. The cursor created is readied for use by an OPEN request, and used by a FETCH request to retrieve rows from the result table.

data access functions

Defined functions in the SQL workspace that have the same names and provide the same kind of database access as certain auxiliary processor operations, specifically PREP, CALL, OPEN, FETCH, CLOSE.

data definition statement

SQL statements that create, alter, or delete a table or view.

data manipulation statement

SQL statements that INSERT, UPDATE, or DELETE data in a table.

defined operator - UNTIL

Defined operator in the SQL workspace used by the workspace functions to create and process a stack of requests to the auxiliary processor.

derived function

A function that is the result of applying an operator to one or two operands in its domain.

error indicator

Last two items of the return code vector; the first of these two indicates where the error was detected; the second gives a diagnostic message number or SQL return code.

field

Intersection of a row and column in a table.

function

An operation that takes zero, one, or two arrays as explicit arguments, and may produce an array as a result.

homogeneous array

A simple array that contains only characters or numbers, but not both.

indicator variable

Variable used in other host languages (*not* APL2) to return information about the existence of NULL values.

isolation level

SQL option that controls the duration of locks held on SQL tables.

Logical Unit of Work

A sequence of SQL commands that are treated by the database system as a single entity. They ensure the consistency of data by guaranteeing that either *all* the data changes made during a logical unit of work are performed, or *none* of them are performed.

marker (character)

A colon required before a vector index in search conditions of an SQL statement submitted through a PREP request.

matrix

An array with rank 2.

mixed array

A simple array that contains both characters and numbers. Contrast with HOMOGENEOUS ARRAY.

NULL

An SQL item with no value, to be distinguished from an empty array in APL2. Null SQL values are represented by (␣0) or (' ') in the data result.

operand

A function or array parameter that is passed to an operator (to be distinguished from an argument of a function).

operation code

Character code assigned to a shared variable to indicate that a particular APL2/SQL interface facility is to be invoked.

operations (auxiliary processor)

Facilities of the APL2/SQL interface that allow direct, shared variable communications with the database system.

options-list

Items passed on a FETCH request or through a SETOPT request to specify the structure of the result data array, the number of rows to be returned, and whether a length matrix is to be returned.

operator

An operation that takes one or two functions and/or arrays as operands and produces a derived function.

primitive functions and operators

Functions and operators whose definitions are built-in to APL2 and need not be defined.

purge

To reset the state of SQL statements to an unprepared state.

query statement

Classification of an SQL SELECT statement by its effect on tables. Only query (SELECT) statements create cursors and return result tables. See also *cursor*.

rank

The number of dimensions of an array.

request

Either an auxiliary processor operation or its equivalent SQL workspace function. See also *operations* and *function*.

request stack (vector)

Vector of auxiliary processor operations created by the SQL function. The operations in the request stack are determined by the SQL statement given as an argument to the SQL function. See also *result data array* and *return code vector*.

result data array

The second item of the result vector returned from an auxiliary processor operation or the SQL function. When the result is returned by a FETCH request on a cursor, the result data array contains a result table. See also *result vector*.

result table

Collection of columns from table(s) created and retrieved using a query (SELECT) statement.

result vector

Two-item vector returned from AP 127 or AP 227 consisting of the return code vector and the result data array.

Three-item vector returned from the SQL function consisting of the return code vector, result data array, and the remainder of the request stack. See also *result data array*.

return code vector

Five-item vector returned to the workspace by AP 127 and AP 227. First three items are the completion code, last two indicate source of error and error diagnosis information.

simple array

An array in which all items are simple scalar characters or numbers, or an empty array of depth 1.

SQL

See *Structured Query Language*.

SQL workspace

APL2 workspace supplied by IBM as part of the APL2/SQL interface. It contains data access, user support, and task control defined functions and the defined operator UNTIL, which provide ways of communicating with AP 127 and AP227.

stack (request)

See *request stack (vector)*.

state

Indication of the status of a particular SQL statement as it is processed using a PREP, CALL or PREP, OPEN, FETCH, CLOSE request sequence; states are *prepared* (ready for CALL), *cursor* (ready for OPEN or DESCRIBE), and *open* (ready for FETCH or CLOSE request).

structure

The arrangement and type of items in an array.

Structured Query Language (SQL)

A nonprocedural language designed to be embedded in procedural languages to provide access to relational databases.

task control functions

Defined functions in the SQL workspace that provide the means to manage the APL2/SQL interface environment, specifically OFFER, COMMIT, CONNECT, ROLLBACK, SETOPT, and GETOPT.

Unit of Recovery

A unit of recovery refers to the period of time that starts when the program starts or the COMMIT or ROLLBACK statement is issued, and ends when a COMMIT, ROLLBACK, or program termination occurs. Any data changes made during this period of time are treated as a single entity with respect to database consistency.

user support functions

Defined functions in the SQL workspace that examine the SQL statement and other arguments given and create the appropriate sequence of auxiliary processor access operations to submit the statement and its value list, if any, specifically QUE, RESUME, SQL, SHOW, and MESSAGE.

value list

Vector of values passed on a CALL request for data manipulation statements and on an OPEN request for a query statement. The vector is indexed by vector indexes embedded in the SQL statement processed using the previous PREP request. See also *vector indexes*.

vector indexes

Numbers, preceded by a marker (colon), embedded in an SQL query or data manipulation statement processed using a PREP request. The numbers refer to a vector of values passed on the subsequent CALL (for data manipulation statements) or OPEN (for query statements). See also *value list*.