DBMaker

ESQL/C Programmer's Guide

CASEMaker®

CASEMaker Inc./Corporate Headquarters

1680 Civic Center Drive

Santa Clara, CA 95050, U.S.A.

www.casemaker.com

www.casemaker.com/support

Document No. 645049-237125/DBM541-M02282017-ESQL

Publication Date: 2017-02-28

For a description of updated functions that do not appear in this manual, read the file named README.TXT after installing the CASEMaker DBMaker software.

Trademarks

CASEMaker, the CASEMaker logo, and DBMaker are registered trademarks of CASEMaker Inc. Microsoft, MS-DOS, Windows, and Windows NT are registered trademarks of Microsoft Corp. UNIX is a registered trademark of The Open Group. ANSI is a registered trademark of American National Standards Institute, Inc.

Other product names mentioned herein may be trademarks of their respective holders and are mentioned only form information purposes. SQL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

Notices

The software described in this manual is covered by the license agreement supplied with the software.

Contact your dealer for warranty details. Your dealer makes no representations or warranties with respect to the merchantability or fitness of this computer product for any particular purpose. Your dealer is not responsible for any damage caused to this computer product by external forces including sudden shock, excess heat, cold, or humidity, nor for any loss or damage caused by incorrect voltage or incompatible hardware and/or software.

Information in this manual has been carefully checked for reliability; however, no responsibility is assumed for inaccuracies. This manual is subject to change without notice.

Contents

1 Introduction

Welcome to the E*SQL\C User's Guide*. *DBMaker* is a powerful and flexible *SQL Database Management System (DBMS)* that supports an interactive *Structured Query Language (SQL)*, a *Microsoft Open Database Connectivity (ODBC)* compatible interface, and *Embedded SQL* for *C (ESQL/C)*. The unique open architecture and native *ODBC* interface give you the freedom to build custom applications using a wide variety of programming tools or to query databases using existing *ODBC*-compliant applications.

DBMaker is easily scalable from personal single-user databases to distributed enterprise-wide databases. The advanced security, integrity, and reliability features of *DBMaker* ensure the safety of critical data. Extensive cross-platform support permits you to leverage existing hardware, allows for expansion and upgrades to more powerful hardware as your needs grow.

DBMaker provides excellent multimedia handling capabilities to store, search, retrieve, and manipulate all types of multimedia data. *Binary Large Objects (BLOBs)* ensures the integrity of multimedia data by taking full advantage of the advanced security and crash recovery mechanisms included in *DBMaker*. *File Objects (FOs)* manage multimedia data while maintaining the capability to edit individual files in the source application.

This manual includes the basic operations of *ESQL\C* and provides systematic instructions that guide you through the management of a database. The *User's Guide* content is intended for designers and administers of *DBMaker* databases. It will assist those unfamiliar with using *DBMaker*, but have some understanding of how a

relational database works. The user should have some operating systems knowledge working with *Windows* and/or *UNIX* environments. Information in this manual may also be helpful for experienced users for reference purposes.

The manual shows various commands and procedures used in maintaining a database with *ESQL\C*. Although the manual is for *DBMaker* on *Windows* environments, it can perform all functions on a *UNIX* platform. For clarity purposes, portions of sample databases appear through out this manual.

SQL is a dual-mode language. It is both an interactive tool to communicate and access a database, commonly referred as an Interactive *SQL*, and a database programming language used by application programs for database access.

Generally, all major RDBMS provide their own user interface for using *SQL*. For example, *DBMaker* provides dmSQL/C. Users can input *SQL* syntax directly through the tool to access and maintain their database.

For the second mode, most major RDBMS also provide two basic techniques for users to use *SQL* in their program. They are Database API and Embedded *SQL*. DBMaker also comes with a variety of Java Tools. For more information on a particular subject, refer to the *Additional Resources* section that follows and select the appropriate manual.

1.1 Additional Resources

DBMaker provides a complete set of RDBMS manuals in addition to this one. For more information on a particular subject, consult one of the books listed below:

- For an introduction to DBMaker's capabilities and functions, refer to the *DBMaker Tutorial*.
- For more information on designing, administering, and maintaining a DBMaker database, refer to the *Database Administrator's Guide*.
- For more information on DBMaker management, refer to the *JServer Manager User's Guide.*
- For more information on DBMaker configurations, refer to the *JConfiguration Tool Reference.*
- For more information on DBMaker functions, refer to the *JDBA Tool User's Guide*.
- For more information on the DCI COBOL interface tool, refer to the *DCI User's Guid*.
- For more information on the SQL language used in dmSQL, refer to the *SQL Command and Function Reference*.
- For more information on the dmSQL, refer to the *dmSQL User's Guide*.
- For more information on the native ODBC API and JDBC API, refer to the *ODBC Programmer's Guide* and *JDBC Programmer's Guide*.
- For more information on error and warning messages, refer to the *Error and Message Reference*.

1.2 Technical Support

CASEMaker provides thirty days of complimentary email and phone support during the evaluation period. When software is registered, an additional thirty days of support will be included, thus, extending the total support period for software to sixty days. However, CASEMaker will continue to provide email support for any bugs reported after the complimentary support or registered support has expired (free of charges).

Additional support is available beyond the sixty days for most products and may be purchased for twenty percent of the retail price of the product. Please contact sales@casemaker.com for more details and prices.

CASEMaker support contact information for your area (by snail mail, phone, or email) can be located at: www.casemaker.com/support. It is recommended that the current database of FAQ's be searched before contacting CASEMaker support staff.

Please have the following information available when phoning support for a troubleshooting enquiry or include the information with a snail mail or email enquiry:

- Product name and version number
- Registration number
- Registered customer name and address
- Supplier/distributor where product was purchased
- Platform and computer system configuration
- Specific action(s) performed before error(s) occurred
- Error message and number, if any
- Any additional information deemed pertinent

1.3 Document Conventions

This book uses a standard set of typographical conventions for clarity and ease of use. The NOTE, Procedure, Example, and CommandLine conventions also have a second setting used with indentation.

CONVENTION	DESCRIPTION
Italics	Italics indicate placeholders for information that must be supplied, such as user and table names. The word in italics should not be typed, but replaced by the actual name. Italics can also be used to introduce new words, and are occasionally used for emphasis in text.
Boldface	Boldface indicates filenames, database names, table names, column names, user names, and other database schema objects. It is also used to emphasize menu commands in procedural steps.
KEYWORDS	All keywords used by the SQL language appear in uppercase when used in normal paragraph text.
SMALL CAPS	Small capital letters indicate keys on the keyboard. A plus sign (+) between two key names indicates to hold down the first key while pressing the second. A comma (,) between two key names indicates to release the first key before pressing the second key.
NOTE	Contains important information.
➲ **Procedure**	Indicates that procedural steps or sequential items will follow. Many tasks are described using this format to provide a logical sequence of steps for the user to follow
➲ **Example**	Examples are given to clarify descriptions, and commonly include text, as it will appear on the screen.
`Command Line`	Indicates text, as it should appear on a text-delimited screen. This format is commonly used to show input and output for dmSQL commands or the content in the dmconfig.ini file

Table 1-1 Document Conventions Table

2 ESQL Basics

This chapter is an introduction to compiling mixed ESQL and C source programs using the ESQL/C preprocessor. Using *SQL* within an Application exclusively for such programs is possible using an Application Program Interface (API); a set of function calls that submits the *SQL* statements to the DBMS and retrieve query results. *DBMaker* provides the industry standard database API-ODBC (Open Database Connectivity).

Embedded *SQL* (ESQL) uses another approach. *SQL* statements, with some minor changes to form, can be written directly into the source code of an application program of the host programming language. This mixed source code is then precompiled by *SQL*, stored in the database, and host language function calls are generated to execute the stored commands.

You can write C application programs that use ESQL commands to access a DBMS. The *DBMaker* ESQL/C preprocessor prepares the application program containing the *SQL* commands for the C compiler. The preprocessor then converts the *SQL* commands to C statements, with C comments, to perform the database operations.

➲ **To create and run an ESQL/C application program:**

1. Design and write the program with embedded *SQL*.
2. Preprocess the program using the *DBMaker* ESQL/C preprocessor *dmppcc*.
3. Compile and link the program generated by the preprocessor.
4. Execute the program.

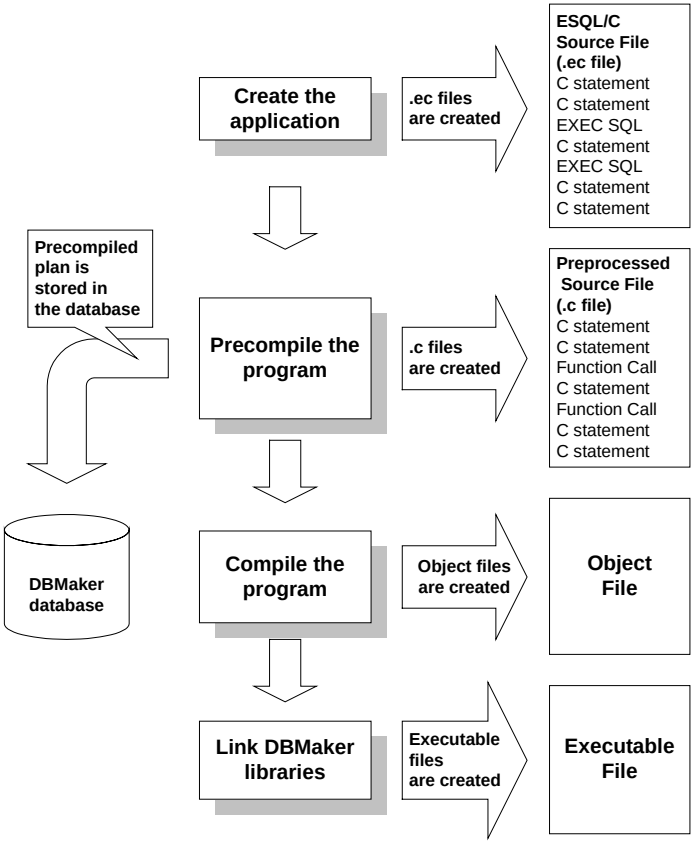

Table 2-1 ESQL program flowchart

2.1 Using dmppcc for Preprocessing

The *DBMaker* preprocessor command is *dmppcc*. The input ESQL/C file has the suffix ".ec" and the output of *dmppcc* is a C language file. During pre-compiling, *dmppcc* creates a stored command for each ESQL DML statement and converts the ESQL statement in the original source to function calls that invoke the stored command.

OPTION	COMMENT
-d database name	Required
-u user name	Required
-p password	Required
-o output_file	The output filename. The default is input_file.c.
-l log filename	The dmppcc error message log. The default is stderr.
-j project name	If not specified, will use module name as project name.
-m module name	If not specified, will use input file name as module name.
-s	Set singleton select error checking on; if not set the default is off.
-cl	Set sql check level = limit. The default is FULL. Users can reference a non-existed table in SQL syntax.
-cs	Set sql check level = syntax. The default is FULL; dmppcc check SQL syntax only, without checking table or column information.
-n	Dmppcc will not store SQL command and execution plan on the database server.
-v	Display version.
-h	Shows help information.
-sp	Compiles the stored procedure source.

Table 2-2 dmppcc *Command Options*

Singleton Select Option

➲ **Example 1**

To use the Singleton Select Option:

```
dmppcc -s ex1.ec
```

➲ **Example 2**

To turn on the Singleton Select Option and confirm run time checking while preprocessing:

```
EXEC SQL SELECT salary FROM emp_table WHERE emp_id = 1000 INTO :var_salary;
```

If singleton select error checking is on, and if the number of the resulting tuple is more than one, then it will return an error.

➲ **Example 3**

Returned error:

```
singleton SELECT can only retrieve at most one row
```

If this check is not turned on, the singleton select query will only retrieve the first tuple column into the host parameter and not check for more resulting tuples.

SQLCHECK

When the SQLCHECK option is set to LIMIT (set -cl), it enables the ESQL preprocessor to continue preprocessing the ESQL/C program without returning an error if a table in the EXEC *SQL* statement does not exist.

When the SQLCHECK option is set to SYNTAX (set -cs), it enables the ESQL preprocessor to continue preprocessing the ESQL/C program without returning an error when a user gives the correct *SQL* syntax in the EXEC SQL statement.

It will not check the related semantic information on the *SQL* statement for the table, column, or security. When this option check is on, dmppcc won't connect to a database, but the user still has to provide the database name for dmppcc to check the database related information in *DBMaker*'s dmconfig.ini file.

When this option is on, dmppcc will not store a precompiled plan in the database. The dmppcc will still check for syntax errors. This option is mainly for multiple user environments, which have multiple persons preprocessing different files that have the same link to an executable file and are testing the executable file with a function that has been preprocessed again but not linked. Users may receive an-- "executable maybe out of date, please rebuild it" error.

Use this option if you receive this type of error message and if you are sure that other people are preprocessing the file or if you receive a "lock timeout" error message.

➲ **Syntax**

```
FULL(default)/LIMIT(-cl)/SYNTAX(-cs)
```

Mandatory Pre-compiling Parameters

➲ **Example 1**

A list of the options that must be set when pre-compiling the ESQL/C program:

```
dmppcc -d test_db -u db_user_id -p db_user_passwd esql_source.ec
```

Where, test_db –u = database name, db_user_id =user name, and db_user_passwd = user's password. These parameters must be provided in the command line or *dmppcc* will try to open and search a local *dmppc.ini* file. You can also put *dmppcc* parameters in the *dmppc.ini* file.

➲ **Example 2**

Syntax in a dmppcc.ini file:

```
DATABASE = database_name
USER = user_id
PASSWORD = password
SELECT_ERROR = yes/no
OUTFILE = output_filename
LOGFILE = log_filename
PROJECT = project_name
MODULE  = module_name
SQLCHECK = FULL/LIMIT/SYNTAX
```

The ".c" file created by *dmppcc* is an ordinary C source code file. You can combine other " .c" or " .o" files to create the final executable file.

➲ **Example 3**

To use a pre-source program ex1.ec and access the database "TESTDB" with the user "john" and the password "johnspwd".

```
dmppcc -d TESTDB -u john -p johnspwd example.ec
```

➲ **Example 4**

To use the C compiler to compile the output file ex1.c, and to link it with the ESQL library and other *DBMaker* libraries as an executable:

```
cc -I. -I$INCDIR -c example.c
```

If the file example.c is preprocessed by the *dmppcc* compiler and compiled to example.o, we may link it with other object files as an executable.

➲ **Example 5**

To link –o to other object files as an executable.

```
acc -o driver example.o otherap.o  -L$LIBDIR -ldmapic -lm
```

You can also reference the Makefile in the *DBMaker*'s samples directory:

~DBMaker/$VERSION/samples/ESQLC.

3 ESQL Syntax

The ESQL/C preprocessor will preprocess all statements that are prefixed by "EXEC *SQL*" or "$" in the ESQL source program. An *SQL* statement can be placed anywhere in a C application program. However, the ESQL preprocessor cannot handle the EXEC SQL statement in a macro definition, and the ESQL/C pre-processor will not preprocess EXEC SQL statements in header files. An *SQL* statement must be preceded by "EXEC SQL" or "$", both keywords must be placed on the same line and end with a semicolon (;).

➲ **Example**

```
if (c1 > 0) EXEC SQL COMMIT WORK;
```

3.1 Static/Dynamic Syntax

If the SQL statement is known at preprocessing time, the ESQL program uses static ESQL syntax. When performing a delete, insert, update, or select operation and the referenced table and search conditions are known then only the input parameter value may be changed at execution time. For this kind of syntax, *DBMaker* will check security, compile it into an execution plan, and then store the plan in the database.

If the *SQL* statement is unknown at the time of writing and preprocessing it is known as dynamic ESQL. Information on how to manage stored information in a database will be covered in Chapter 7,*Project and Module Management*.

The application can use dynamic ESQL syntax when the complete or partial *SQL* statement is unknown at preprocessing time. The *SQL* statement is given by the user, (e.g. dmsqlc), and the whole *SQL* statement is composed by the query tool (QBE). *DBMaker* will not be able to compile the *SQL* syntax at preprocessing time for dynamic ESQL syntax. Therefore, all compilation and security check operations are executed at run time. Detailed dynamic ESQL syntax will be illustrated in Chapter 6, *Dynamic ESQL*.

When a host variable is referenced in a general C statement, the method is the same as for other C variables. When the host variable is referenced in an EXEC SQL statement, it must start with a colon (:).

➲ **Example 1**

```
EXEC SQL INCLUDE DBENVCA;
EXEC SQL INCLUDE SQLCA;
main ( )
    {
     EXEC SQL BEGIN DECLARE SECTION;
     char sql_string[255];
     EXEC SQL END DECLARE SECTION;

     EXEC SQL CONNECT TO testdb john johnspwd;
```

```
    /* get user input for SQL statement                                    */
    user_input(sql_string);
    /* Dynamic ESQL statement without host variable                        */
    EXEC SQL PREPARE statement_name FROM :sql_string;
    EXEC SQL EXECUTE statement_name;
    EXEC SQL DISCONNECT;
   }
```

➲ **Example 2**

Static ESQL syntax:

```
EXEC SQL CONNECT TO database_name user_name password
EXEC SQL DISCONNECT [database_name]
EXEC SQL INCLUDE {DBENVCA | SQLCA | SQLDA}
EXEC SQL WHENEVER {SQLERROR | SQLWARNING | NOT FOUND}
                  {CONTINUE| STOP | GO TO label| GOTO label | DO action}
EXEC SQL BEGIN DECLARE SECTION, EXEC SQL END DECLARE SECTION
EXEC SQL [AT DATABASE_NAME] any SQL statement
EXEC SQL [AT DATABASE_NAME] DECLARE cursor_name CURSOR FOR sql_query_statement
EXEC SQL [AT DATABASE_NAME] OPEN cursor_name [USING host_variable_[indicator]_list]
EXEC SQL [AT  DATABASE_NAME] FETCH cursor_name [INTO host_variable_[indicator]_list]
EXEC SQL [AT DATABASE_NAME] CLOSE cursor_name
```

➲ **Example 3**

Static ESQL/C program format:

```
EXEC SQL INCLUDE DBENVCA;
EXEC SQL INCLUDE SQLCA;
main ( )
    {
     EXEC SQL BEGIN DECLARE SECTION;
     int  emp_id;
     char emp_name[20], emp_addr[50];
     EXEC SQL END DECLARE SECTION;
     EXEC SQL CONNECT TO testdb john johnspwd;
```

```
    /* get user input for host var                                    */
    user_input(&emp_id, emp_name, emp_addr);
    EXEC SQL INSERT INTO emp_table VALUES (:emp_id, :emp_name, :emp_addr);
    EXEC SQL DISCONNECT;
}
```

3.2 Variables

You can use variables called *host variables* from the host program in ESQL statements to pass data between the C application and the *DBMaker* database. A host variable is always accompanied by another variable called an indicator variable. While the host variable holds a value, the *indicator variable* registers the special nature of that value if it is NULL or has been truncated.

Declare Section

Any host variable and indicator variable that have been referenced in EXEC *SQL* statements must be in the declare section. The declare section is made up of C variable declarations contained within the EXEC *SQL* statements, BEGIN DECLARE SECTION, and END DECLARE SECTION.

There can be any number of declare sections within an application program. Every function should have its own declare section if it has host variables and indicator variables referenced in the EXEC *SQL* statement. *DBMaker*'s *dmppcc* will return an error if the host variable and indicator variable cannot be found in the declare section.

➲ **Example**

A BEGIN DECLARE SECTION:

```
EXEC SQL BEGIN DECLARE SECTION;
varchar hoEmpNo[8];                          /* A host variable       */
int    inEmpNo;                              /* An indicator variable */
EXEC SQL END DECLARE SECTION;
```

Host Variable Data Types

A singleton C variable can be declared in a host variable. C structures and unions are not allowed in a host variable, except for one-dimensional character arrays that are used to specify char buffer length. The fileobj data type cannot be used for specifying a one-dimensional array. Other types of C arrays can be declared as a one dimensional

array for fetching a row set with more than one data value in a single FETCH statement. The user can use a two dimensional array to retrieve more than one data value in a single FETCH statement for CHAR or BINARY types.

ESQL/C TYPE	TYPE DEFINITION	DEFINITION
char var_name[n]	char[n]	fix length char input (n) char output (n-1) char+ NULL terminate
binary var_name[n]	char[n]	fix length binary input (n) char output (n) char
short	short	short integer
int	int	integer
long	long	long integer
float	float	float
double	double	double

Table 3-1 ESQL types defined in esqltype.h

The following *SQL* data types can also be used in the declare section:

ESQL/C TYPE	TYPE DEFINITION	DEFINITION
date	typedef struct date_s { short year; unsigned short month; unsigned short day;} eq_date;	Date
time	typedef struct time_s { unsigned short hour; unsigned short minute; unsigned short second; } eq_time;	Time

ESQL/C Type	Type Definition	Definition
timestamp	typedef struct timestamp_s { short year; unsigned short month; unsigned short day; unsigned short hour; unsigned short minute; unsigned short second; unsigned long fraction;} eq_timestamp;	Timestamp
varchar var_name[n]	typedef struct varchar_s {long len; char arr[n]; } varchar;	Variable length char. Input: Null terminated string if user has not specified len. Output: (n-1) char +Null terminated.
varcptr var_name[n]	typedef struct varcptr_s {long len; char *arr; } varcptr;	Variable length char, user must assign the len value. Input/output same as varchar
varbinary var_name[n]	typedef struct varbinary_s {long len;char arr[n];} varbinary;	Variable length binary, user must assign the len value. Input/output same as binary.
varbptr	typedef struct varbpr_s { long len; /* must assign a buffer length */ char *arr; /* must assign a valid buffer ptr address */ } varbptr;	Variable length binary Input/output: same as varbinary

Table 3-2 Declare Section SQL *data types*

The **varchar** type is a null terminated variable length character string, and **varbinary** is a variable length binary string without null terminate. Vary the input or output variable's length by assigning the actual length in the **varchar** or **varbinary** type's len field. The buffer length and buffer addresses are not defined for the **varcptr** or **varbptr** type; remember to assign them before using them in the program.

For these non-standard C data types, *dmppcc* converts them into C structures that are recognized by the C compiler. For example, varchar is resolved into a C structure having a field length and character array elements.

BLOB DATA TYPE	TYPE DEFINITION	DEFINITION
Longvarchar	typedef struct longvarchar_s { long bufsize; char *buf; } longvarchar;	BLOB (text data)
Longvarbinary	typedef struct longvarbinary_s { long bufsize; char *buf; } longvarbinary;	BLOB (binary data)
fileobj	typedef struct fileobj_s { long type; char fname[MAX_FNAME_LEN]; } fileobj;	BLOB (file object) Default type is ESQL_STORE_FILE_CONTENT. You can also set type as ESQL_STORE_FILE_NAME to store only file name on server.

Table 3-3Blob Data Types

FILEOBJ'S

If the file type is not set then the default type is ESQL_STORE_FILE_CONTENT, and the database server will store the user specified file content. It will not matter if the file has been deleted after you have inserted the file.

However, if you set file type = ESQL_STORE_FILE_NAME, the database will only store the file name specified in the fname field; you must make sure that the file is accessible by *DBMaker*'s database server. If you delete the file, *DBMaker* will return an error when you reference it. The settings of file type will only work when fileobj type is used as an input parameter. As an output parameter, the database will always try to copy the data into the file specified.

➲ **Example 1**

Fileobj type as an input host variable:

```
EXEC SQL BEGIN DECLARE SECTION;
fileobj fname1;
EXEC SQL END DECLARE SECTION;
EXEC SQL CREATE TABLE t1 (c1 file);
strcpy(fname1.name , "u:\image_path\test1.gif" );
/* This INSERT statement will store all the content of file test.gif   */
EXEC SQL INSERT INTO t1 VALUES (:fname1);
fname1.type = ESQL_STORE_FILE_NAME;
strcpy(fname1.name , "u:\image_path\test2.gif");
EXEC SQL INSERT INTO t1 VALUES (:fname1);
```

➲ **Example 2**

Fileobj type as an output variable with the schema for table t2 as c1 long varchar:

```
EXEC SQL BEGIN DECLARE SECTION;
fileobj fname1;
EXEC SQL END DECLARE SECTION;
strcpy(fname1.name , "u:\image_path\test1.gif");
/* This SELECT statement will fetch the blob data from server site and put into user's
local file.   */
strcpy(fname1.name , "u:\local_path\test1.gif");
```

```
EXEC SQL select c1 from t2 into :fname1;
```

➲ **Example 3**

Fileobj as an output variable using cursor:

```
EXEC SQL BEGIN DECLARE SECTION;
fileobj fname1;
EXEC SQL END DECLARE SECTION;
int idx1=0;
strcpy(fname1.name , "u:\image_path\test1.gif");
EXEC SQL DECLARE myCur1 CURSOR FOR select c1 from t2 into :fname1;
While (1)
{
idx1++;
/* This FETCH statement will fetch the blob data from server site and store it into
user's local file.  If you do not change output file name, in the next FETCH statement,
the output file  will be overwritten. */
sprintf(fname1.name , "test%d.gif", idx1);
EXEC SQL FETCH myCur1;
if (SQLCODE)
  { /* Break while loop when no more data or there's error */
  if (SQLCODE != SQL_SUCCESS_WITH_INFO)
         break;
  }
}
```

Host Variables

Once host variables are established, the data can be stored in the variable by *DBMaker* for use by the application program. A host variable can also be used to insert and update data, or in the WHERE and HAVING clauses.

Apply the following when using host variables in a C application program:

- Declare the host variables according to the regular syntax of the *C* language and DBMaker's ESQL supported data types.

- The *INTO* clause host variables is equal to or less than the number of columns named in the *SELECT* statement and the data type of the input or output host variable is a compatible data type corresponding with the parameter or projection column.
- Use host variables compatible with column value types.
- Use host variables prefixed by a colon *(:)* in SQL statements; when used elsewhere in an application program no colon is necessary.

➲ **Example 1**

```
EXEC SQL BEGIN DECLARE SECTION;
char emp_id[20];                                /* employee ID          */
char emp_tel[20];                               /* employee telephone #  */
int indvalue = SQL_NTS;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT emp_id, emp_tel FROM emp_tab INTO :emp_id :indvalue,
:emp_tel :indvalue;
```

➲ **Example 2**

```
EXEC SQL BEGIN DECLARE SECTION;
char emp_id[20];                                /* employee ID          */
char emp_tel[20];                               /* employee telephone #  */
int indvalue = SQL_NTS;  /* indicates input parameter is null terminated */
EXEC SQL END DECLARE SECTION;
strcpy(emp_id, "john Smith");
strcpy(emp_tel, "765-4321");
EXEC SQL INSERT INTO emp_tab VALUES (:emp_id :indvalue, :emp_tel :indvalue);
```

➲ **Example 3**

To use output host variable hoDeptNo and input host variable hoEmpNo, initialized with the code for select criteria.

```
EXEC SQL BEGIN DECLARE SECTION;
int  hoDeptNo, hoEmpNo;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT deptNo FROM Employee WHERE empNo = :hoEmpNo INTO :hoDeptNo;
```

Variable Scope

Declare the variable scope as EXTERN or STATIC in the declare section in the same way as other C variables.

➲ **Example 1**

To reference an external (Global) variable with a supported ESQL data type, add "extern" in front of the ESQL variable declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
extern int var1;
EXEC SQL END DECLARE SECTION;
```

➲ **Example 2**

To set the local (Static) variable as a static variable, add "static" in front of the ESQL variable declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
static int var1;
EXEC SQL END DECLARE SECTION;
```

➲ **Example 3**

To reference the value from the function's input variable, or to return the value obtained from an ESQL or *SQL* query statement, copy the variable data or reassign the pointer to the ESQL variable's data structure.

```
/* Method 1, copy the value to esql host variable */
func1(int input_emp_id, char *output_telno)
    {
     EXEC SQL BEGIN DECLARE SECTION;
     int  emp_id;
     char telno[15];
     EXEC SQL END DECLARE SECTION;
     emp_id = input_emp_id;                  /* copy the input value */
     EXEC SQL SELECT telno FROM emp_tab WHERE emp_id = :emp_id INTO :telno;
     strcpy(output_telno, telno);
    }
```

```
/* Method 2, reassign the buffer pointer */
func1(char *input_emp_name, char *output_telno)
    {
     EXEC SQL BEGIN DECLARE SECTION;
     varcptr p_name, p_telno;
     EXEC SQL END DECLARE SECTION;
     p_name.len  = strlen(input_emp_name);   /* input string length  */
     p_name.arr  = input_emp_name;
     p_telno.len = 15;                       /* output string length */
     p_telno.arr = output_telno;
     EXEC SQL SELECT telno FROM emp_tab WHERE emp_name = :p_name INTO :p_telno;
    }
```

Indicator Variables

Indicator variables are an optional means to handle null values and truncation for host variables in the application program. When an indicator variable is used, it follows a host variable in the *SQL* statements. Declare indicator variables are integers only.

The indicator variable indDeptNo, immediately follows the host variable hoDeptNo and is prefixed with a colon.

➲ **Example 1**

Declare indicator variable:

```
EXEC SQL BEGIN DECLARE SECTION;
int  hoDeptNo, indDeptNo, hoEmpNo;
EXEC SQL END DECLARE SECTION;
EXEC SQL SELECT deptNo FROM Employee
                      WHERE empNo = :hoEmpNo
                      INTO :hoDeptNo :indDeptNo;
```

Indicator Value	Value Retrieved From the Database
SQL_NULL_DATA	NULL value returned; output host variable is indeterminate.
0 or greater	The indicator variable will be set as the original output host variable buffer length. If the indicator variable is for getting BLOB data, the indicator value set to the original output host variable buffer length in the database prior getting the blob data.

Table 3-4 Example 1Indicator Variable Values

➲ **Example 2**

Use an indicator variable to input a NULL value in the database; the corresponding host variable is ignored.

```
EXEC SQL BEGIN DECLARE SECTION;
int  hoDeptNo, indDeptNo;
EXEC SQL END DECLARE SECTION;
inDeptNo = SQL_NULL_DATA;
EXEC SQL INSERT INTO Department (DeptName, DeptNo)
             VALUES ('Human Resource', :hoDeptNo :inDeptNo);
```

The above example will insert the NULL value into the DeptNo column, and the value of hoDeptNo will be ignored.

Indicator Value	Value Accepted Into the Database
SQL_NULL_DATA	NULL value.
0 or greater	Actual buffer length of input host variable, for CHAR and BINARY types. When an indicator variable is provided, the host variable length set in VARCHAR/VARBINARY data types is ignored.
SQL_NTS	Buffer is null terminated for CHAR and BINARY data types. The host variable length set in VARCHAR/VARBINARY data types is ignored.

Table 3-5 Example 2 Indicator Variable Values

3.3 Status Codes

There are three kinds of declarations for special data structures required by *DBMaker*'s preprocessor or the runtime application that can be used with the Include Variable. The syntax for declaring sqlca and sqlda are the same as dbenvca. Sqlca is used to communicate the status code and sqlda is used in dynamic ESQL.

- dbenvca
- sqlca
- sqlda

dbenvca

The dbenvca declaration is an environment variable *DBMaker* uses in an application program; it must be declared in the program.

➲ **Syntax**

```
EXEC SQL INCLUDE [EXTERN] dbenvca;
```

➲ **Example**

```
file1.c
EXEC SQL INCLUDE DBENVCA;
main()
    {
     ...
     EXEC SQL .
     ...
    }
file2.c
EXEC SQL INCLUDE EXTERN DBENVCA;
func1()
    {
     ...
```

```
}
```

SQLCA

Status codes for each executed *SQL* command are returned into the *SQL* Communication Area (SQLCA). *DBMaker* uses variables contained in this data structure to pass status information to the C program, where the information can be analyzed and handled if any problems arise.

DECLARING SQLCA

The SQLCA structure variable must be globally accessible in ESQL/C programs. SQLCA and DBENVCA are both global variables that must be declared in the ESQL/C source program. SQLCA is a structure declared automatically by the preprocessor when it encounters the following statement.

➲ **Example**

To automatically declare SQLCA:

```
EXEC SQL INCLUDE [EXTERN] SQLCA;
```

STATUS RETURNED IN SQLCA

Status information from the database server is returned through the SQLCA. It is the application program's responsibility to analyze the data and handle errors or warnings.

There are two ways to instruct the application program to examine the status codes in the SQLCA and handle errors and warnings. You can write the commands for this in C code or use the *SQL*, WHENEVER command, to generate error handling during C code preprocessing.

➲ **Example**

SQLCA syntax definition:

```
#define MAX_ERR_STR_LEN 256
/*-------------------------------------------------------------
* SQLCA - the SQL Communications Area (SQLCA)
*-------------------------------------------------------------*/
```

```
typedef struct sqlca
    {
     unsigned char  sqlcaid[8];          /* the string "SQLCA   "     */
     long           sqlcabc;             /* length of SQLCA, in bytes */
     long           sqlcode;             /* SQLstatus code            */
     long           sqlerrml;            /* length of sqlerrmc data   */
     unsigned char  sqlerrmc[MAX_ERR_STR_LEN]; /* name of object cause error */
     unsigned char  sqlerrp[8];          /* diagnostic information    */
     long           sqlerrd[6];          /* various count and error code */
     unsigned char  sqlwarn[8];          /* warning flag array          */
     unsigned char  sqlext[8];           /* extension to sqlwarn array   */
    } sqlca_t;
#define SQLCODE   sqlca.sqlcode   /* SQL status code  */
#define SQLWARN0 sqlca.sqlwarn[0] /* master warning flag */
#define SQLWARN1 sqlca.sqlwarn[1] /* string truncated */
```

DBMaker's error, warning, and other codes are stored in sqlca.sqlerrd[0]. The number of fetched rows is stored at sqlca.sqlerrd[3]. You can reference them when fetching more than one row in a FETCH statement. For more information, refer to the *DBMaker Error and Message Reference Guide*.

SQLCODE	MEANING
SQL_SUCCESS or 0	Success
SQL_ERROR or (-1)	Error
SQL_NO_DATA_FOUND or 100	No rows satisfied the search condition
SQL_SUCCESS_WITH_INFO or 1	Warning

Table 3-6 Sample Codes

3.4 The WHENEVER Statement

The WHENEVER statement can be used to handle errors. When the ESQL/C preprocessor encounters the WHENEVER statement, it generates C error handling code with execution dependent on the outcome of an *SQL* statement.

The default value of each condition is CONTINUE. A WHENEVER statement affects all *SQL* statements that come after it in an application program, up to the next WHENEVER for the same condition. In other words, the most recent setting of the WHENEVER statement remains in effect for all of the following EXEC *SQL* statements to the end of the file, unless another WHENEVER statement in the middle overrides it.

To prevent the possibility of an infinite loop, do not forget to set WHENEVER *check_case* CONTINUE in the error handler, or in any other functions containing an EXEC *SQL* statement that requires error handling using the WHENEVER statement.

Use these conditions in the WHENEVER statement to direct an application program:

- *STOP* - rolls back your work, disconnects from the database, and terminates the application program when an SQL statement's return status meets the specified condition.
- *CONTINUE* - disables the condition set in the previous *WHENEVER* statement and continues execution with the statement next to the one that caused the error.
- *GOTO label_name* - directs execution to the label for an error-handling routine within your application program.
- *DO c_action_statement* - Does a specified action when the returned status meets the specified condition.

CONDITION	DESCRIPTION
SQLERROR	When SQLCODE is SQL_ERROR
SQLWARNING	When SQLCODE is SQL_SUCCESS_WITH_INFO
NOT FOUND	When SQLCODE is SQL_NO_DATA_FOUND

Table 3-7 Sample Codes

➲ **Example 1**

```
WHENEVER SQLERROR DO break;
WHENEVER SQLWARNING DO print_warning();
 while ()
{
EXEC SQL …..;
EXEC SQL….;
}
```

➲ **Example 2**

```
int func()
    {
     ...
     EXEC SQL WHENEVER SQLERROR goto error_handle;
     EXEC SQL INSERT INTO emp_table VALUES (:emp_id, :emp_name, :emp_addr);
     ...
     ...
     return 0;
error_handle:
     EXEC SQL WHENEVER SQLERROR CONTINUE;
     printf("ERR:%s\n", sqlcode.sqlerrmc);
    return -1;
    }
```

4 Data Manipulation

In this chapter, we will give some examples of using ESQL in an application, including how to use host variables, indicator variables, and NULL values. You can issue the same *SQL* statements within an ESQL application that are available in interactive *SQL*, plus some additional ones described in this manual.

All *SQL* commands can be prefixed by EXEC *SQL* in an ESQL application, including: COMMIT, ROLLBACK, CONNECT, DISCONNECT, INSERT, SELECT, UPDATE, DELETE, etc. However, it is not common to use DDL *SQL* statements in an ESQL application.

Examples of additional embedded *SQL* statements used in ESQL are declarative statements such as BEGIN (END) DECLARE SECTION, DECLARE CURSOR, INCLUDE, and WHENEVER as well as additional executable statements such as CLOSE, DESCRIBE, EXECUTE (IMMEDIATE), FETCH, OPEN, and PREPARE. These executable statements are used in embedded *SQL* only.

4.1 Data Manipulation

Only host variables are used to pass data from the application to the database for INSERT, UPDATE, and DELETE. In addition to declaring the host variables in the declare section, initialize every input host variable before referencing it.

The keyword in *SQL* syntax for testing the NULL value in the WHERE clause, 'IS NULL', cannot use an indicator variable to indicate the NULL value in the WHERE clause.

➲ **Example 1**

```
EXEC SQL BEGIN DECLARE SECTION;
int   hoDeptNo, inDeptNo;
varchar  hoDeptName[8];
EXEC SQL BEGIN DECLARE SECTION;
/* Use host variable hoDeptNo to input data into database              */
hoDeptNo = 1001;
EXEC SQL INSERT INTO Department (DeptName, DeptNo)
             VALUES ('Human Resource', :hoDeptNo);
```

➲ **Example 2**

Input the NULL value in the database with the corresponding host variable ignored:

```
inDeptNo = SQL_NULL_DATA;
EXEC SQL INSERT INTO Department (DeptName, DeptNo)
             VALUES ('Human Resource', :hoDeptNo :inDeptNo);
```

➲ **Example 3**

To use the host variable of *SQL* pseudo data types such as varchar, which are not directly supported by the C language:

```
strcpy(hoDeptName.arr, 'Human Resource');
hoDeptName.len = strlen(hoDeptName.arr);
hoDeptNo = 1001;
EXEC SQL INSERT INTO Department (DeptName, DeptNo)
```

```
                VALUES (:hoDeptName, :hoDeptNo);
```

➲ **Example 4**

To use the UPDATE and DELETE input host variables:

```
strcpy(hoDeptName.arr, 'Human Resource');
hoDeptName.len = strlen(hoDeptName.arr);
hoDeptNo = 1001;
EXEC SQL UPDATE Department SET DeptNo = :hoDeptNo WHERE DeptName = :hoDeptName;
EXEC SQL DELETE FROM Department WHERE DeptNo = :hoDeptNo + 1;
```

4.2 Retrieving Single-Row Data

Retrieve single-row data from the database to the output host variable, passing a value to the application. Use output host variables in the INTO clause of a SELECT statement.

In the example following, hoDeptName is an output host variable and hoDeptNo is an input host variable. An indicator variable inDeptName has also been appended to the output host variable to check whether the query retrieves the NULL value or a truncated value. The 0 value indicates a normal result.

SQL_NULL_DATA indicates the host variable received a NULL value. A value greater than 0 indicates that the result in the host variable is truncated and the value in the indicator variable is the length of the original value.

For example, since the length of hoDeptName is 8, the result in the database has a length of 12, the indicator variable will be 12 and the host variable will contain the first 8 characters of the original value.

You can have the full range of standard *SQL* clauses (WHERE, GROUP BY, ORDER BY, HAVING, etc.), within the embedded SELECT statement. You can use input host variables in the WHERE clause and HAVING clause.

Every SELECT statement retrieves exactly one row, if a query returns more than one row use a cursor (see below). Check the sqlca.sqlcode after each SELECT statement to see how many rows have been retrieved. If the only sqlcode in SQLCA is SQL_NO_DATA_FOUND, no data was found. This option has to be specified in *DBMaker*'s preprocessor *dmppcc* to return an error when more than one row is retrieved with the SELECT statement.

➲ **Example 1**

```
EXEC SQL BEGIN DECLARE SECTION;
int   hoDeptNo, inDeptNo, inDeptName;
varchar  hoDeptName[8];
EXEC SQL BEGIN DECLARE SECTION;
```

```
hoDeptNo = 1001;
EXEC SQL SELECT DeptName FROM Department
          WHERE DeptNo = :hoDeptNo
           INTO :hoDeptName :inDeptName;
```

➲ **Example 2**

If this option is set, an error will be shown when the result is more than one row.

```
dmppcc -d TESTDB -u john -p johnspwd -s ex1.ec
```

4.3 Transaction Processing

Use the EXEC SQL COMMIT and EXEC SQL ROLLBACK commands to control the integrity of a transaction. For more advanced applications use SAVEPOINT and ROLLBACK TO SAVEPOINT options to have better control over data processing. COMMIT WORK and ROLLBACK WORK erase all of the Savepoints set in a transaction. If you exit the program without issuing COMMIT or ROLLBACK, all changes in the transaction will be rolled back.

You can set the AUTOCOMMIT connection option in the dmconfig.ini file. If the AUTOCOMMIT connection option is set on, then all of the *SQL* statements executed are committed immediately and the ESQL application cannot perform the rollback statement. Be sure to turn the AutoCommit option off before running the application or use the following syntax in an ESQL source file to turn on/off the AutoCommit option after connecting to a database.

➲ **Example**

To use the SET AUTOCOMMIT ON/OFF:

```
EXEC SQL SET AUTOCOMMIT {ON|OFF}
```

4.4 Dynamic connection syntax

Sometimes you might want to access multiple databases with an ESQL/C application. Write several ESQL programs, preprocess them with different database names, and link all programs as an executable or add the "AT database name" in front of the *SQL* statement, and specify the database names before executing the *SQL* statement.

➲ **Example**

```
EXEC SQL BEGIN DECLARE SECTION;
 char db1[20];
 char usr1[10];
 char pwd1[10];
 int c1;
EXEC SQL END DECLARE SECTION;
/* GetDBInfo ()is user function which will pass back database name, user, password */
GetDBInfo(db1, usr1, pwd1);
EXEC SQL CONNECT TO :db1 :usr1 :pwd1;
EXEC SQL AT :db1 select c1 from t1 into :c1;
EXEC SQL DISCONNECT :db1;
```

4.5 Using a Cursor

When a query returns more than one row, the program must execute the query differently. Multiple-row queries are handled in two stages. A program starts the query and no data is returned immediately. Then, the program requests the data rows one at a time via a cursor.

A cursor, as used in an application program, is a data selector that can operate on specified rows from the database. The following operations are performed with the DECLARE, OPEN, FETCH, and CLOSE statements.

➲ **To use a cursor**

1. Allocate storage to hold the cursor, declare the cursor and its associated SELECT statement.
2. Start the execution of the associated SELECT statement and open the cursor.

   NOTE: It actually contains three sub-steps, namely parse the SQL statements, bind the host variables, and begin execution of the statement.
3. Fetch, delete, or update a row of data into host variables and process it.

   NOTE: Repeat this step until all rows have been fetched.
4. Close the cursor.

Declaring a Cursor

Declare a new cursor for each SELECT command within an application program, except when retrieving a single row.

The INTO host_variable clause can also be defined in the FETCH statement. If you want to use all available methods of fetching data (i.e., FETCH NEXT, PREVIOUS, LAST, FIRST, ABSOLUTE, and RELATIVE), specify SCROLL when declaring a cursor.

➲ **Example 1**

The DECLARE syntax:

```
EXEC SQL DECLARE cursor_name [SCROLL] CURSOR FOR select_statement
        [INTO :output_host_var :indicator_var [, :output_host_var :indicator_var]]
```

➲ **Example 2**

To fetch more than one row in a single FETCH statement, specify the SCROLL keyword.

```
EXEC SQL DECLARE vendCursor SCROLL CURSOR FOR
          SELECT vendorName FROM vendors
           WHERE vendorNumber = :inputNo;
```

➲ **Example 3**

Alternatively use the INTO clause in the DECLARE statement.

```
EXEC SQL DECLARE vendCursor CURSOR FOR
          SELECT vendorName FROM vendors
           WHERE vendorNumber = :inputNo
            INTO :vendName;
```

Opening a Cursor

A cursor must be in an open state to operate on the contents of the specified rows. The OPEN command is followed by the name of the cursor given in the DECLARE command.

When the OPEN command is executed, the cursor finds and points to the first row of the result set that satisfies the search condition. If there are input host variables in the cursor, you must specify the values for all input host variables before or in the OPEN cursor statement. In the above case, because the input host variables have been defined in the DECLARE statement, there's no need to specify the input host variables again in the OPEN cursor statement.

➲ **Example 1**

The OPEN syntax:

```
EXEC SQL OPEN cursor_name
       [USING :input_host_var [:indicator_var]
           [, :input_host_var [:indicator_var]]]
```

➲ **Example 2**

The *SQL* command for opening a cursor

```
EXEC SQL OPEN vendCursor;
```

Using a Cursor to Retrieve Data

A cursor retrieves data by using the FETCH command. In a loop, FETCH advances the cursor in the result set and retrieves the current row, copying the values of the columns specified in the SELECT list into the host variables designated by the INTO clause.

The INTO clause can be omitted in the FETCH statement. Where nth_position and num_rows can be a host variable or a constant integer. PREVIOUS, FIRST, LAST, ABSOLUTE nth_position, RELATIVE nth_position and num_rows ROWS are available only with cursors defined with the SCROLL option.

You must use a FETCH command for each row to be retrieved. After all rows in the result set have been retrieved, *DBMaker* sets the SQLCODE field of SQLCA to the value SQL_NO_DATA_FOUND to indicate that no more rows are found. Indicated variables can be declared with the host variables to detect null values.

When the, num_rows ROW, syntax is specified in a FETCH statement or in an INTO arrayed host variable, you can retrieve more than one data value in a FETCH statement.

➲ **Example 1**

The FETCH syntax:

```
EXEC SQL FETCH [NEXT | PREVIOUS | FIRST | LAST | ABSOLUTE nth_position | RELATIVE
nth_position] [num_rows ROWS]
        cursor_name
        [INTO :input_host_var [:indicator_var]
              [, :input_host_var [:indicator_var], ...]]
```

➲ **Example 2**

An *SQL* command for fetching results from a cursor:

```
EXEC SQL FETCH vendCursor INTO :vendName;
```

➲ **Example 3**

The FETCH ROWS command:

```
EXEC SQL BEGIN DECLARE SETION;
int nrows;
int host1[50];
int host2[50];
char host3[50][100]; /* 50: means the maximum available fetched data values,100: means
the maximum data buffer length of each element. */
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE myCur SCROLL CURSOR FOR SELECT c1,c2,c3 FROM table1
INTO :host1, :host2, :host3;
EXEC SQL OPEN myCur;
nrows = 50;
/* loop until no data found */
while (SQLCODE != SQL_NO_DATA_FOUND)
{
     EXEC SQL FETCH :nrows ROWS myCur;  /* This will fetch 50 rows into host1, host2,
host3, because the maximum fetched  */
```

```
    for (j = 0; j < sqlca.sqlerrd[3]; j++) /* print out according to the number of
returned rows */
     printout(host1, host2, host3);
}
EXEC SQL CLOSE myCur;
```

➲ **Example 4**

Returns the next row within the results set. NEXT is the default cursor fetch.

```
EXEC SQL FETCH cur1;            /* default is fetch next */
EXEC SQL FETCH NEXT cur1 INTO :c1, :c2;
```

➲ **Example 5**

The PREVIOUS command returns the previous row within the results set.

```
EXEC SQL FETCH PREVIOUS cur1 INTO :c1, :c2;
```

➲ **Example 6**

The FIRST command moves the cursor to the first row within the result set and returns the first row.

```
EXEC SQL FETCH FIRST cur1 INTO :c1, :c2;
```

➲ **Example 7**

The LAST command moves the cursor to the last row within the result set and returns the last row.

```
EXEC SQL FETCH LAST cur1 INTO :c1, :c2;
```

➲ **Example 8**

The ABSOLUTE n command returns the *n* th row within the results set. If *n* is a negative value, the returned row will be the *n* th row counting backward from the last row of the results set.

```
n = 10;
EXEC SQL FETCH ABSOLUTE :n cur1 INTO :c1, :c2;
```

➲ **Example 9**

The RELATIVE n command returns the *nth* row after the currently fetched row. If *n* is a negative value, the returned row will be the nth row counting backward from the relative position of the cursor.

```
n = 10;
EXEC SQL FETCH RELATIVE :n cur1 INTO :c1, :c2;
```

Deleting Data with a Cursor

With a cursor, you can select one row at a time to be deleted from a table. Any additional rows must be fetched individually for deletion.

Do not use the COMMIT WORK command between consecutive deletions. Executing this command closes the cursor and terminates the deletion process. Working with AUTOCOMMIT mode on will have the same results. Turn AUTOCOMMIT mode off before using a DELETE or UPDATE WHERE CURRENT OF cursor statement.

➲ **To delete a row**

1. Declare the cursor with a SELECT command.
2. Use the OPEN and FETCH commands to open the cursor and position it on the row to be deleted.
3. Execute the DELETE command.
4. To delete another row, reposition the cursor with another FETCH command.

➲ **Example**

```
EXEC SQL DELETE FROM supplier WHERE CURRENT OF vendCursor;
```

Updating Data with a Cursor

You can use a cursor to select one row at a time to update. Reposition the cursor to update additional rows.

Do not use the COMMIT WORK command between consecutive updates, and make sure AUTOCOMMIT mode is off. Either of these will close the cursor and terminate the update process.

➲ **To update a row**

1. Declare the cursor with a SELECT command.
2. Use the OPEN and FETCH commands to open the cursor and position it on the row to be updated.
3. Execute the UPDATE command.
4. To update another row, reposition the cursor with another FETCH command.

➲ **Example**

```
EXEC SQL UPDATE supplier SET price = price + 10
          WHERE CURRENT OF vendCursor;
```

Closing the Cursor

COMMIT WORK and ROLLBACK WORK commands will implicitly close a cursor. You can explicitly close a cursor with the CLOSE CURSOR command. When you no longer need a cursor, you should close it to free any allocated resources. The cursor is dropped and cannot be used, opened or fetched, again.

➲ **Example 1**

The CLOSE CURSOR syntax:

```
EXEC SQL CLOSE cursor_name
```

➲ **Example 2**

An *SQL* command for closing a cursor:

```
EXEC SQL CLOSE vendCursor;
```

5 BLOB Data

When the BLOB data size is unknown at preparation time, or a buffer is not large enough, you may want to PUT or GET partial BLOB data each time until the entire data has been retrieved.

Use the original ESQL syntax to retrieve the BLOB column if you can allocate enough buffer size in a program or do not care whether you can GET the whole BLOB or not.

Because it is not necessary to GET BLOB data in sequence, we can assign a column to GET. Using the GET BLOB statement, data does not have to be retrieved from left column to right column, (small column numbers to large column numbers). It is also not necessary to GET the entire BLOB data if it is not required.

PUT BLOB must start from the first to the last BLOB column and indicated when to begin and stop. If you did not PUT each BLOB column accordingly or indicate when to end the PUT BLOB then the BLOB column will not be inserted in the database.

Therefore, the operation will be aborted, because it never finished, when you disconnect or exit from the program. If you disconnect from the database, an error will show that the last statement has been canceled.

5.1 PUT BLOB Statement

➲ **To use a PUT BLOB statement**

1. Prepare an ESQL statement and declare the BLOB host variable as a question mark, to indicate that this BLOB host variable will be bound later.

The PREPARE syntax is:

```
EXEC SQL PREPARE stmt_name FROM "SQL SYNTAX"
| :sql_string_host_variable];
```

dmppcc will treat the "SQL SYNTAX" as static ESQL syntax. The syntax will be parsed and the execution plan will be stored when preprocessing.

dmppcc will treat PREPARE as Dynamic ESQL syntax when it includes the :sql_string_host_variable,. The syntax will not be parsed until run time. Refer to the Chapter on *Dynamic ESQL* for more details.

If you want to handle a BLOB with GET or PUT BLOB syntax later, define it using " ? " as the BLOB host variable.

To insert emp_pic into emp_table:

```
EXEC SQL PREPARE stmt1 FROM

        "INSERT INTO emp_table (emp_id, emp_pic) VALUES
(:emp_id, ?)";
```

2. Execute this ESQL statement:

```
EXEC SQL EXECUTE stmt_name;
```

Example

```
EXEC SQL EXECUTE stmt1;
```

3. Declare when to BEGIN to PUT the BLOB:

```
EXEC SQL BEGIN PUT BLOB FOR stmt_name;
```

Example

```
EXEC SQL BEGIN PUT BLOB FOR stmt1;
```

4. Define which BLOB variable to use for putting the BLOB and filling the information of the bufsize and bufptr. The order for PUT BLOB must be sequential from the first to the last unbound BLOB column.

```
EXEC SQL PUT BLOB FOR stmt_name USING :host_var [:indicator_var]

/*host_var's data type must be longvarchar or longvarbinary)      */
```

Example

```
strcpy(buf, "This is a test");

b1.buf     = buf;

b1.bufsize = strlen(buf);

EXEC SQL PUT BLOB FOR stmt1 USING :b1;
```

5. Follow step 4 until there is no more BLOB data to PUT in this column.

6. If there is more than one BLOB column to PUT, declare when to begin putting the next BLOB column and return to step 4.

```
EXEC SQL PUT NEXT BLOB FOR stmt_name;
```

Example

```
EXEC SQL PUT NEXT BLOB FOR stmt1;
```

7. If the entire BLOB column has been PUT into the database, declare that the PUT BLOB operation is finished.

```
EXEC SQL END PUT BLOB FOR stmt_name;
```

Example

```
EXEC SQL END PUT BLOB FOR stmt1;
```

➲ **Syntax**

The entire PUT BLOB statement:

```
 * Prepare an insert statement; the input BLOB columns should use a
 * question mark to denote it.
*****************************************************************/
EXEC SQL PREPARE stmt1 FROM "insert into emp_table \
             (emp_id, emp_pic, emp_memo) values (:id, ?, ?)";
```

```
id = 1000+j;
/*****************************************************************
* Execute this statement.
*****************************************************************/
EXEC SQL execute stmt1;

/*****************************************************************
* If there are 300 characters to put into emp_pic, and we want to
* put it 100 characters at a time.
*****************************************************************/
pic_buffer.bufsize  = 100;         /* max buffer size                */
/* you must allocate enough buffer as indicated in field bufsize, or
*point to a valid buffer pointer */
pic_buffer.buf  = user_buf;
/*****************************************************************
* Begin PUT BLOB.
*****************************************************************/
EXEC SQL BEGIN PUT BLOB FOR stmt1;
/*****************************************************************
* Loop 3 times to put data.
*****************************************************************/
for (i=0; i < 3; i++)
    {
     sprintf(pic_buffer.buf, "user's picture %dth's data........ ", i);
     EXEC SQL PUT BLOB FOR stmt1 USING :pic_buffer;
    }

/*****************************************************************
* Now start to put the next BLOB column's data.
*****************************************************************/
EXEC SQL put next blob FOR stmt1;
/*****************************************************************
* If there are 200 characters to put into emp_memo, and we want to
* put it 100 characters at a time.
```

```
*************************************************************/
memo_buffer.bufsize  = 100;         /* max buffer size               */
/* you must allocate enough buffer as indicated in field bufsize, or
*point to a valid buffer pointer */
memo_buffer.buf  = user_buf;
/*************************************************************
* loop 2 times to put data
*************************************************************/
for (i=0; i < 2; i++)
    {
     sprintf(memo_buffer.buf, "user's memo %dth's data..........", i);
     EXEC SQL PUT BLOB FOR stmt1 USING :memo_buffer;
    }
/*************************************************************
* end PUT BLOB
*************************************************************/
EXEC SQL END PUT BLOB FOR stmt1;
```

5.2 GET BLOB Statement

➲ **To use the GET BLOB statement**

1. Prepare an ESQL statement and declare the BLOB host variable as a question mark (it means this BLOB host variable is not yet bound).

```
EXEC SQL PREPARE stmt_name FROM "SQL SYNTAX";
```

Unlike dynamic ESQL syntax, the *SQL* SYNTAX is known at preprocessing time, and must include the bound host variable name; if it's a BLOB, and you want to use the GET/PUT BLOB method, then you must define '?'.

Suppose we want to fetch emp_pic from emp_table:

Example

```
EXEC SQL PREPARE stmt1 FROM "select emp_pic from emp_table into ?";
```

2. Declare a cursor for a prepared statement.

```
EXEC SQL DECLARE cursor_name CURSOR FOR stmt_name;
```

Example

```
EXEC SQL DECLARE myCur CURSOR FOR stmt1;
```

3. Open the cursor.

Example

```
EXEC SQL OPEN myCur;
```

4. Fetch the cursor.

Example

```
EXEC SQL FETCH myCur;
```

5. You do not need to declare when to begin a GET BLOB, but you must define the column number in the BLOB host variable and the available buffer size and valid buffer pointer.

```
EXEC SQL GET BLOB COLUMN :blobcol_num  FOR stmt_name USING :host_var
[:indicator_var]
```

Syntax 1

```
EXEC SQL BEGIN DECLARE SECTION;

int nCol;

longvarchar b1;

EXEC SQL END DECLARE SECTION;

nCol = 1;         /* assign nCol as the column order in projection
*/

b1.bufsize = 50;  /* if you want to get 50 bytes at a time        */

b1.buf  = buf;    /* assign a valid buffer pointer to b1          */

EXEC SQL GET BLOB COLUMN :nCol FOR stmt2 USING :b1
```

NOTE: You can specify an indicator variable to GET the remaining buffer size before getting the BLOB column.

Syntax 2

```
EXEC SQL PREPARE stmt1 FROM "select c6 from d1 into ?";

EXEC SQL EXECUTE stmt1;

/* fetch BLOB with size = 0 first, to know total size with indicator
*/

nCol    = 1;

b1.bufsize = 0;

b1.buf     = wkbuf;

EXEC SQL GET BLOB COLUMN :nCol FOR stmt1 USING :b1 :i_b1;

if (SQLCODE == 0)

     printf("left size is %d\n", i_b1);

/* loop fetch BLOB with size = 2, and print out   "last" remain size
*/

for (i = 0; SQLCODE != SQL_NO_DATA_FOUND; i++)
```

```
{
 b1.bufsize = 2;       /* get 1 char plus null ptr at a time */
 EXEC SQL GET BLOB COLUMN :nCol FOR stmt1 USING :b1 :i_b1;
 if (SQLCODE != SQL_NO_DATA_FOUND)
      printf("b1 = %s, last remain size is %d\n", b1.buf, i_b1);
 chkErr();
}
```

NOTE: You can also set bufsize = DB_ALLOCATE_MEMORY, when you want DBMaker to allocate the memory for retrieving the BLOB. DBMaker will free the allocated memory related to the BLOB when you call next FETCH or CLOSE CURSOR statement. Use caution with this option, because by allocating enough memory for the BLOB may cause an "OUT OF MEMORY" problem in the system. In addition, since the memory is freed by the database after the next FETCH or CLOSE statement, the pointer of the BLOB variable will refer to an invalid address. This may cause a core dump or error execution result in your program.

6. Continue as in the previous step, if there is more BLOB data to GET.

Example

```
EXEC SQL PREPARE stmt1 FROM "select c6 from d1 into ?";
EXEC SQL EXECUTE stmt1;
/* fetch BLOB with size = 0 first, to know total size with indicator
*/
nCol    = 1;
b1.bufsize = 0;
b1.buf     = wkbuf;
EXEC SQL GET BLOB COLUMN :nCol FOR stmt1 USING :b1 :i_b1;
```

```
if (SQLCODE == 0)
     printf("left size is %d\n", i_b1);
/* loop fetch BLOB with size = 2, and print out   "last" remain size
*/
for (i = 0; SQLCODE != SQL_NO_DATA_FOUND; i++)
    {
     b1.bufsize = 2;       /* get 1 char plus null ptr at a time */
     EXEC SQL GET BLOB COLUMN :nCol FOR stmt1 USING :b1 :i_b1;
     if (SQLCODE != SQL_NO_DATA_FOUND)
          printf("b1 = %s, last remain size is %d\n", b1.buf, i_b1);
     chkErr();
    }
```

7. If all data has been retrieved or you do not want to GET the rest of the BLOB data, you can continue to FETCH the rest of the result buffer for the cursor until no more rows are found.

Example

```
#define MAX_BUF_SIZE 101
/* Prepare a SELECT statement, the output BLOB column should use    */
/* a question mark to denote it.                                    */
EXEC SQL PREPARE stmt1 FROM "select emp_id, emp_pic from emp_table
                                 into :id, ?";
/* Declare a cursor to associate it with this statement.            */
EXEC SQL DECLARE myCur CURSOR FOR stmt1;
/* Open this cursor.                                                */
EXEC SQL OPEN myCur;
```

```
/* To get one MAX_BUF_SIZE character at a time, we must fill the
* following field first.                                          */
nCol = 2;                  /* second output column in projection.*/
pic_buffer.bufsize  = MAX_BUF_SIZE;   /* max buffer size          */
/* You must allocate enough buffer as indicated in field bufsize, or
   point to a valid buffer pointer.                               */
pic_buffer.buf  = user_buf;
/* loop fetch result.                                             */
while (1)
    {
     EXEC SQL FETCH myCur INTO :id, ?; /* fetch cursor            */
     if (SQLCODE)
         {
          if (SQLWARN0 != 'W')
              break;
         }
     printf("emp_id = %d\n", id);
     printf("emp_pic = ");
     /* loop get BLOB data until no data is found.                */
     for (i = 0; SQLCODE != SQL_NO_DATA_FOUND; i++)
         {
          EXEC SQL GET BLOB COLUMN :nCol FOR stmt1
                              USING :pic_buffer :pic_ind;
          if (SQLCODE != SQL_NO_DATA_FOUND)
```

```
            {
             if (pic_ind == SQL_NULL_DATA)
                 printf("(null)");
             else
                 {
                  for(j = 0; j < MAX_BUF_SIZE-1; j++)
                  printf("%c", pic_buffer.buf[j]);
                 }
            }
        }
     printf("\n");
    }

/* Close the cursor.                                          */
EXEC SQL CLOSE myCur;
```

6 Dynamic ESQL

You can write an ESQL statement in two ways. The simpler and more common way is by static embedding, which means that the *SQL* statement is written as part of the source program text before pre-compiling. Up to this point, we have exclusively covered Static ESQL.

Although static ESQL is extremely useful, it requires you to know the exact syntax for *SQL* statements at the time of writing a program. Some applications require the ability to compose *SQL* statements in response to user input at run time. This can be done with dynamic ESQL, in which the program composes an *SQL* statement as a string of characters and passes it to the database at run time. All or part of the dynamic ESQL statement is unknown at the precompiled time; the complete statement is constructed in memory during run time.

The dynamic ESQL statements are categorize into four types according to whether it is a SELECT statement and whether it has known or unknown host variables. Each type of dynamic ESQL uses a different method to program the ESQL statement.

TYPE	CHARACTERISTIC	METHOD
1	Non query, no input host variable	Execute Immediate
2	Non query, known number of input host variables	Prepare/Execute
3	Query, known number of input and output host variables	Cursor
4	Unknown number of input or output host variables	SQLDA

Table 6-1ESQL Statement Catagories

6.1 Type 1 Dynamic ESQL

Type 1 dynamic ESQL is a non-SELECT statement without input host variables. In this simple case, you can use EXECUTE IMMEDIATE to process the dynamic ESQL.

➲ **Example**

Type 1 Dynamic ESQL:

```
EXEC SQL BEGIN DECLARE SECTION;
varchar upd_str[100];
EXEC SQL END DECLARE SECTION;
sprintf(upd_str.arr, "UPDATE part SET qty = qty -1 WHERE ");
gets (update_condition);                    /* get dynamic upd condition */
strcat (upd_str.arr, update_condition);     /* construct dynamic SQL     */
upd_str.len = strlen(upd_str.arr);

EXEC SQL EXECUTE IMMEDIATE FROM :upd_str;   /* execute it                */
EXEC SQL COMMIT WORK;
```

6.2 Type 2 Dynamic ESQL

Type 2 dynamic ESQL is a little bit more complicated. It is a non-SELECT statement with a known number of input host variables. If the number of input host variables is undetermined at precompiled time, then you must use type 4 dynamic ESQL (see the section later in this chapter).

➲ **To construct a type 2 Dynamic ESQL application**

1. Declare all input host variables in the declare section.
2. Prepare this statement:

```
EXEC SQL PREPARE statement_name FROM :statement_string;
```

3. Set value of all input host and indicate variables.
4. Execute this statement:

```
EXEC SQL EXECUTE statement_name USING :input_var1, :input_var2;
```

➲ **Example**

To construct a type 2 Dynamic ESQL application:

```
EXEC SQL BEGIN DECLARE SECTION;
varchar del_str[80];
int  ord_number;
EXEC SQL END DECLARE SECTION;
char del_condition[80];
/* there is an input variable :iVord, it is a place holder              */
sprintf(del_str.arr, "DELETE FROM order WHERE ordid = :iVord AND ");
gets (del_condition);               /* get the DYNAMIC delete condition */
strcat (del_str.arr, del_condition); /* construct dynamic SQL           */
del_str.len = strlen(del_str.arr);
EXEC SQL PREPARE del_stmt FROM :del_str;
/* please note the relationship between the input host                  */
/* variable ord_number and placeholder iVord.                         */
gets (ord_number);                  /* set host variable value         */
```

```
EXEC SQL EXECUTE del_stmt USING :ord_number;
EXEC SQL COMMIT WORK;
```

6.3 Type 3 Dynamic ESQL

Type 3 dynamic ESQL is a SELECT statement with a known number of input and output host variables. If the number of either input or output host variables is undetermined at precompiled time, it becomes type 4 dynamic ESQL.

➲ **Processing type 3 dynamic ESQL**

1. Prepare a statement string inside a host variable, the statement string including a SQL statement with a placeholder '?' for each input variable.
2. Execute an ESQL PREPARE statement specifying the statement name and statement string.

```
EXEC SQL PREPARE statement_name FROM :statement_string;
```

3. Execute the ESQL DECLARE CURSOR statement specifying the cursor name and the statement name.

```
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name;
```

4. Specify a value for each input variable.
5. Open the cursor with a list of input variables.
6. In a loop, fetch the result to a list of output variables.
7. Close the cursor.

➲ **Example**

Processing type 3 dynamic ESQL:

```
EXEC SQL BEGIN DECLARE SECTION;
varchar sel_str[100];
int  ord_num, ord_date, custor_num;
EXEC SQL END DECLARE SECTION;
/* 1. Put a build statement string inside a host variable, including one */
/*    placeholder for each input variable.                               */
gets(condition);
sprintf(sel_str.arr, "SELECT Ordid, Orddate FROM order WHERE CusId = :c \
                        AND %s", condition);
```

```
sel_str.len = strlen(sel_str.arr);
/* 2. Prepare the statement.                                          */
EXEC SQL PREPARE sel_stmt FROM :sel_str;
/* 3. Declare the cursor.                                             */
EXEC SQL DECLARE emp_cursor CURSOR FOR sel_stmt;
/* 4. Specify a value for each input variable.                        */
gets (custor_num);
/* 5. Open the cursor with a list of input variables.                 */
EXEC SQL OPEN emp_cursor USING :custor_num;
/* 6. In a loop, fetch the result to a list of output variables.      */
do
    {
     EXEC SQL FETCH emp_cursor INTO :ord_num, :ord_date;
     printf("ord_num = %d ord_date = %d\n", ord_num, ord_date);
    } while (sqlca.sqlcode == SQL_SUCCESS ||
             sqlca.sqlcode == SQL_SUCCESS_WITH_INFO)
/* 7. Close the cursor.                                               */
EXEC SQL CLOSE emp_cursor;
```

6.4 Type 4 Dynamic ESQL

Type 4 dynamic ESQL is an *SQL* statement with input or output host variables undefined at precompiled time. With this type of dynamic ESQL, the SQLDA descriptor, host variables must be used. There are many steps for type 4, however in certain situations some of the steps may be skipped. Like, the number of output host variables is unknown but the number of input host variables is known, or vice versa.

SQLDA Descriptor

An SQLDA descriptor is an area in which the application and *DBMaker* store the number, value, length, data type, and name of each host variable and indicates the variable value in the dynamic ESQL statement.

SQLDA has the information for the number of host variables and the description of the host variables. The number of host variables equals the number of input or output host variables involved in the current *SQL* statement. The description information has two sets of information: one contains the column information, and the other contains the host variable information.

Describe Command

Type 4 dynamic ESQL does not know the number of columns that are involved in the *SQL* statement until the user inputs the statement at run time. Hence, the application does not know how many input or output host variables are needed to pass information in and out of the database.

This is why we use a descriptor area to store host variable information in SQLDA. After the dynamic ESQL is prepared, the application program should use the Describe command to ask *DBMaker* how many columns are there for input (DESCRIBE BIND VARIABLE) and how many output columns are there for the output (DESCRIBE SELECT LIST) and what they are.

The Describe command will put the number of columns (i.e., host variables) and the data of these columns into a descriptor. In a loop, the application should check what

kind of input or output columns are there, and then allocate space for the host variables.

Passing information through SQLDA

DBMaker supports two functions for allocating and freeing SQLDA:

```
allocate_descriptor_storage()
free_descriptor_storage()
```

If any error occurs, information will also be stored in SQLCA.

➲ **Prototype 1**

```
int allocate_descriptor_storage(long maxNumber, char **descriptor_name);
```

➲ **Example 1**

To allocate an SQLDA descriptor, 'desc1', with a maximum of 10:

```
char  *desc1;
long   maxNumber = 10;
/* SQLCODE is macro of sqlca.code                                    */
/* support error_handle() is a function for error handling           */
allocate_descriptor_storage(maxNumber, &desc1);
if (SQLCODE == SQL_ERROR) error_handle();
```

➲ **Example 2**

To free an SQLDA descriptor, 'desc1':

```
free_descriptor_storage(desc1);
if (SQLCODE == SQL_ERROR) error_handle();
```

SQLDA contains host variable and column information. DBMaker sets the column information during the description time. The host variable information is set by the application. DBMaker or the application sets the indicator variable information. The application can use the function SetSQLDA() to set host variable information and use the function GetSQLDA() to get column information. If any error occurs, information will also be stored in SQLCA.

➲ **Prototype 2**

The SetSQLDA prototype command is:

```
int SetSQLDA(char *descriptor_name, short host_variable_number,
            short option, long option_value);
```

➲ **Prototype 3**

The GetSQLDA prototype command is:

```
int GetSQLDA(char *descriptor_name, short projection_column_number,
            short option, void *option_value);
```

OPTION	DESCRIPTION
descriptor_name	An SQLDA descriptor allocated by allocate_descriptor_storage().
Host variable number	The number of host variable
Projection column number	The number of project column list
Option	An option symbol
Option value	A valid option value

Table 6-2 Function Arguments:

When option is SQLDA_NUM_OF_HV or SQLDA_MAX_FETCH_ROWS, host_variable_number or projection_column_number is not used. Option symbols are used to specify what kind of information you will SET or GET.

OPTION	DESCRIPTION
SQLDA_DATABUF	Specifies that the option_value is the data buffer pointer.
SQLDA_DATABUF_LEN	Specifies that the option_value is the data buffer maximum length.
SQLDA_DATABUF_TYPE	Specifies that the option_value is the data type of the data buffer.
SQLDA_BLOB_FLAG	Specifies that the host variable will be handled by BLOB method

OPTION	DESCRIPTION
SQLDA_PUT_DATA_LEN	Specifies that the option_value is the length of the host variable put data.
SQLDA_INDICATOR	Specifies that the option_value is the indicator value.
SQLDA_MAX_FETCH_ROWS	Specifies that the option_value is the maximum number of fetched rows
SQLDA_STORE_FILE_TYPE	Specifies option_value is type of store file (ESQL_STORE_FILE_CONTENT or ESQL_STORE_FILE_NAME)
SQLDA_COLTYPE	Specifies that the option_value is the column type of the host variable.
SQLDA_COLLEN	Specifies that the option_value is the column length of the host variable.
SQLDA_COLPREC	Specifies that the option_value is the column precision of the host variable.
SQLDA_COLSCALE	Specifies that the option_value is the column scale of the host variable.
SQLDA_COLNULLABLE	Specifies that the option_value is a column host variable that can contain null values.
SQLDA_COLNAME	Specifies that the option_value is the column name of the host variable.
SQLDA_COLNAME_LEN	Specifies that the option_value is the column name length of the host variable.
SQLDA_NUM_OF_HV	Specifies that the option_value is the total number of host variables involved in this current SQL statement.

Table 6-3 SET and GET Options

OPTION	OPTION VALUES
SQLDA_NUM_OF_HV	Integer 0 ~ 252

OPTION	OPTION VALUES
SQLDA_MAX_FETCH_ROWS	Positive integer
SQLDA_DATABUF	Valid pointer
SQLDA_DATABUF_LEN	Positive integer
SQLDA_DATABUF_TYPE	See following 'Data Type of data buffer table
SQLDA_STORE_FILE_TYPE	ESQL_STORE_FILE_CONTENT ESQL_STORE_FILE_NAME
SQLDA_COLTYPE	See following 'Data Type of column' table
SQLDA_COLLEN	Positive integer depend on value of SQLDA_COLTYPE
SQLDA_COLPREC	Positive integer depend on value of SQLDA_COLTYPE
SQLDA_COLSCALE	Positive integer depend on value of SQLDA_COLTYPE
SQLDA_COLNULLABLE	SQL_NO_NULLS – column is not nullable SQL_NULLABLE – column is nullable
SQLDA_COLNAME	Character string
SQLDA_COLNAMELEN	Integer 1 ~ 18
SQLDA_INDICATOR	SQL_NULL_DATA – input NULL data SQL_DEFAULT_PARAM – input DEFAULT value SQL_NTS – input data until null terminate Positive integer – real length of input data
SQLDA_BLOB_FLAG	SQLDA_BLOB_ON SQLDA_BLOB_OFF
SQLDA_PUT_DATA_LEN	Positive integer

Table 6-4 Data Types and Option Values

The value of the SQLDA_DATABUF_TYPE will tell *DBMaker* the type of data in the data buffer specified by SQLDA_DATABUF.

SQLDA_DATABUF_TYPE	C TYPE OF SQLDA_DATABUF
SQL_C_CHAR	Character pointer (printable character string)
SQL_C_LONG	long
SQL_C_SHORT	Short
SQL_C_FLOAT	Float
SQL_C_DOUBLE	Double
SQL_C_BINARY	Character pointer (not printable character string)
SQL_C_DATE	ESQL define type 'eq_date' (see esqltype.h)
SQL_C_TIME	ESQL define type 'eq_time' (see esqltype.h)
SQL_C_TIMESTAMP	ESQL define type 'eq_timestamp' (see esqltyep.h)
SQL_C_FILE	Character string (value of SQLDA_DATABUF is a file name)

Table 6-5 Buffer Data Types

SQLDA_COLTYPE	CORRESPONDING DATA TYPE OF COLUMN
SQL_CHAR	char
SQL_VARCHAR	varchar
SQL_LONGVARCHAR	Long varchar
SQL_BINARY	binary
SQL_LONGVARBINARY	Long varbinary
SQL_INTEGER	int
SQL_SMALLINT	smallint
SQL_REAL	float
SQL_DOUBLE	double
SQL_DECIMAL	decimal

SQLDA_COLTYPE	CORRESPONDING DATA TYPE OF COLUMN
SQL_DATE	date
SQL_TIME	time
SQL_TIMESTAMP	timestamp
SQL_FILE	file

Table 6-6 Column Data Types

You can get the values of all referenced options using GetSQLDA().

Only some options can be set:

- SQLDA_COLTYPE
- SQLDA_COLLEN
- SQLDA_COLPREC
- SQLDA_COLSCALE
- SQLDA_COLNULLABLE
- SQLDA_COLNAME
- SQLDA_COLNAME_LEN
- SQLDA_NUM_OF_HV

The following tables give detailed information about who will set the SQLDA options and when they are set. Remember the application first uses DESCRIBE to ask for column information from *DBMaker*. *DBMaker* sets the column information. The application then sets host variable information and will ask *DBMaker* to execute the statement.

OPTION SYMBOL	SET BY	WHEN SET
SQLDA_NUM_OF_HV	DBMaker	during DESCRIBE BIND VARIABLES
SQLDA_DATABUF	application	before OPEN/EXECUTE
SQLDA_DATABUF_LEN	application	before OPEN/EXECUTE

Option Symbol	Set by	When Set
SQLDA_DATABUF_TYPE	application	before OPEN/EXECUTE
SQLDA_STORE_FILE_TYPE	application	before OPEN/EXECUTE
SQLDA_COLTYPE	DBMaker	During DESCRIBE BIND VARIABLES
SQLDA_COLLEN	DBMaker	during DESCRIBE BIND VARIABLES
SQLDA_COLPREC	DBMaker	during DESCRIBE BIND VARIABLES
SQLDA_COLSCALE	DBMaker	during DESCRIBE BIND VARIABLES
SQLDA_COLNULLABLE	DBMaker	during DESCRIBE BIND VARIABLES
SQLDA_COLNAME	Not used	--
SQLDA_COLNAME_LEN	Not used	--
SQLDA_INDICATOR	application	before OPEN/EXECUTE
SQLDA_MAX_FETCH_ROWS	application	before open
SQLDA_BLOB_FLAG	Application	before OPEN/EXECUTE
SQLDA_PUT_DATA_LEN	application	before PUT BLOB

Table 6-7 Input Host Variables

Descriptor Fields	Set by	When Set
SQLDA_NUM_OF_HV	DBMaker	during DESCRIBE SELECT LIST
SQLDA_DATABUF	Application	before FETCH
SQLDA_DATABUF_LEN	Application	before FETCH
SQLDA_DATABUF_TYPE	Application	before FETCH
SQLDA_STORE_FILE_TYPE	Not used	during DESCRIBE SELECT LIST

DESCRIPTOR FIELDS	SET BY	WHEN SET
SQLDA_COLTYPE	DBMaker	during DESCRIBE SELECTLIST
SQLDA_COLLEN	DBMaker	during DESCRIBE SELECT LIST
SQLDA_COLPREC	DBMaker	during DESCRIBE SELECT LIST
SQLDA_COLSCALE	DBMaker	during DESCRIBE SELECT LIST
SQLDA_COLNULLABLE	DBMaker	during DESCRIBE SELECT LIST
SQLDA_COLNAME	DBMaker	during DESCRIBE SELECT LIST
SQLDA_COLNAME_LEN	DBMaker	during DESCRIBE SELECT LIST
SQLDA_INDICATOR	DBMaker	during FETCH
SQLDA_BLOB_FLAG	application	before FETCH
SQLDA_PUT_DATA_LEN	Not used	--

Table 6-8 Output Host variables

Application Steps

There are many steps in constructing a type 4 dynamic ESQL application. Some of the steps may be omitted if you know there are no input or output host variables.

➲ **To construct a type 4 dynamic ESQL application**

1. Declare descriptor variables.
2. Allocate SQLDA for dynamic input/output host variables by maxNumber.
3. Execute an SQL PREPARE statement specifying the statement name and statement string.

```
EXEC SQL PREPARE statement_name FROM :statement_string;
```

4. Declare a cursor for the statement prepared in step 3.

```
EXEC SQL DECLARE cursor_name CURSOR FOR statement_name; // step 4 and
5 for input host variable.
```

NOTE: You only need to go through step 5 when there are input host variables.

5. Describe the input host variables in the statement prepared in step 3 and put the information into the bound descriptor.

```
EXEC SQL DESCRIBE BIND VARIABLES FOR statement_name INTO
descriptor_name;
```

a) Set the length of input host variables.

b) Set the data type of input host variables.

c) Allocate storage for the value of input host variables.

d) Set the value of input host variables.

e) Set the value of input indicator variables.

6. Open the cursor you declared in step 4, and specify the descriptor variables the cursor should use.

```
EXEC SQL OPEN cursor_name USING DESCRIPTOR descriptor_name; // step
7 and 8 for output host variable.
```

7. Describe the output column projection in the statement prepared in step 3 and put these describe information into the bound descriptor.

```
EXEC SQL DESCRIBE SELECT LIST FOR statement_name INTO
descriptor_name;
```

8. Set the length of the output host variables, set the data type of output host variables, and allocate storage for the value of output host variables.

9. Fetch data by cursor declared in step 4 and put fetched data into data buffer of bound descriptor.

```
EXEC SQL FETCH cursor_name USING descriptor_name;
```

10. Close the cursor.

```
EXEC SQL CLOSE cursor_name;
```

11. Free user allocated memory space for SQLDA (the pData field in SQLDA).

12. De-allocate the descriptor.

➲ **Example 1**

Type 4 dynamic ESQL application:

```
#define maxNumber 10
#define STRING_LEN 128
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
/* 0. declare descriptor variables                                    */
char *input_descriptor, *select_descriptor;
long  nHv=0, nCol=0;
char *pColName, *pData;
long  colType, colScale, colNullable;
long   colLen, colNameLen, dataType, colPrec;
/* connect to database                                                */
EXEC SQL CONNECT TO :dbname :user :password;
/* 1. allocate SQLDA for dynamic in/out host variables by maxNumber   */
allocate_descriptor_storage(maxNumber, &input_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
allocate_descriptor_storage(maxNumber, &select_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
/* 2. EXEC SQL PREPARE statement_name FROM :statement_string          */
gets(stmt_str.arr);
stmt_str.len = strlen(stmt_str.arr);
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
/* 3. EXEC SQL DECLARE cursor_name CURSOR FOR statement_name          */
EXEC SQL DECLARE demo_cursor CURSOR FOR demo_stmt;
/* 4. EXEC SQL DESCRIBLE BIND VARIABLES FOR statement_name INTO       */
/*    input_descriptor                                                */
EXEC SQL DESCRIBE BIND VARIABLES FOR demo_stmt INTO input_descriptor;
GetSQLDA(input_descriptor, 0, SQLDA_NUM_OF_HV, &nHv);
if (SQLCODE == SQL_ERROR) error_handle();
printf("There are %d returned input host variables: \n\n", nHv);
```

```
/* 5. set length of input host variables, set dataType of input host     */
/*    variables,allocate storage for value of input host variables, set  */
/*    value of input host variables, set value of input indicates, set   */
/*    type, len, allocate buffer, value                                  */
for (i = 1; i <= nHv; i++)
    {
     pData = malloc(STRING_LEN);
     SetSQLDA(input_descriptor, i, SQLDA_DATABUF, pData);
     if (SQLCODE == SQL_ERROR) error_handle();
     SetSQLDA(input_descriptor, i, SQLDA_DATABUF_TYPE, SQL_C_CHAR);
     if (SQLCODE == SQL_ERROR) error_handle();
     strcpy(pData, "dynamic ESQL/C example");
     datalen = strlen(pData);
     SetSQLDA(input_descriptor, i, SQLDA_INDICATOR, datalen);
     if (SQLCODE == SQL_ERROR) error_handle();
    }
/* 6. EXEC SQL OPEN cursor_name USING DESCRIPTOR input_descriptor        */
EXEC SQL OPEN demo_cursor USING DESCRIPTOR input_descriptor;
/* 7. EXEC SQL DESCRIBLE SELECT LIST FOR statement_name INTO             */
/*    select_descriptor                                                  */
EXEC SQL DESCRIBE SELECT LIST FOR demo_stmt INTO select_descriptor;
GetSQLDA(select_descriptor, 0, SQLDA_NUM_OF_HV, &nCol);
if (SQLCODE == SQL_ERROR) goto error_label;
printf("There are %d returned columns: \n\n", nCol);
for (i = 1; i <= nCol; i++)
    {
     printf("column %d : \n");
     GetSQLDA(select_descriptor, i, SQLDA_COLNAME_LEN, &colNameLen);
     if (SQLCODE == SQL_ERROR) error_handle();
     GetSQLDA(select_descriptor, i, SQLDA_COLNAME, &pColName);
     if (SQLCODE == SQL_ERROR) error_handle();
     GetSQLDA(select_descriptor, i, SQLDA_COLTYPE, &colType);
     if (SQLCODE == SQL_ERROR) error_handle();
     GetSQLDA(select_descriptor, i, SQLDA_COLLEN, &colLen);
```

```
    if (SQLCODE == SQL_ERROR) error_handle();
    GetSQLDA(select_descriptor, i, SQLDA_COLPREC, &colPrec);
  if  (SQLCODE == SQL_ERROR) error_handing(); )
    GetSQLDA(select_descriptor, i, SQLDA_COLSCALE, &colScale);
    if (SQLCODE == SQL_ERROR) error_handle();
    GetSQLDA(select_descriptor, i, SQLDA_COLNULLABLE, &colNullable);
    if (SQLCODE == SQL_ERROR) error_handle();
    printf(" column name length = %ld \n", colNameLen);
    printf(" column name = %s \n", pColName);
    printf(" column type = %ld \n", colType);
    printf(" column length = %ld \n", colLen);
    printf(" column precision = %ld \n", colPrec);
    printf(" column scale = %ld \n", colScale);
    printf(" column nullable = %ld \n", colNullable);
   }
/* 8. set length of output host variables, set dataType of output host   */
/* variables, allocate storage for value of output host variable         */
for (i = 1; i <= nCol; i++)
    {
    GetSQLDA(select_descriptor, i, SQLDA_COLTYPE, &colType);
    if (SQLCODE == SQL_ERROR) error_handle();
    GetSQLDA(select_descriptor, i, SQLDA_COLLEN, &colLen);
    if (SQLCODE == SQL_ERROR) error_handle();
    switch (colType)
        {
         case SQL_CHAR:
              pData = malloc(colLen+1);
              SetSQLDA(select_descriptor, i, SQLDA_DATABUF, pData);
              if (SQLCODE == SQL_ERROR) error_handle();
              SetSQLDA(select_descriptor, i, SQLDA_DATABUF_LEN, colLen+1);
              /* '+1' for null terminate                                  */
              if (SQLCODE == SQL_ERROR) error_handle();
              SetSQLDA(select_descriptor,i,SQLDA_DATABUF_TYPE,SQL_C_CHAR);
              if (SQLCODE == SQL_ERROR) error_handle();
```

```
                break;
           case SQL_INTEGER:
                pData = malloc(4);
                SetSQLDA(select_descriptor, i, SQLDA_DATABUF, pData);
                if (SQLCODE == SQL_ERROR) error_handle();
                SetSQLDA(select_descriptor, i, SQLDA_DATABUF_LEN, 4);
                if (SQLCODE == SQL_ERROR) error_handle();
                SetSQLDA(select_descriptor, i, SQLDA_DATABUF_TYPE, SQL_C_LONG);
                if (SQLCODE == SQL_ERROR) error_handle();
                break;
           }
      }
/* 9. EXEC SQL FETCH cursor_name USING select_descriptor                */
while (1)
    {
     EXEC SQL FETCH demo_cursor USING select_descriptor;
if (SQLCODE != SQL_SUCCESS && SQLCODE != SQL_SUCCESS_WITH_INFO)
         break;
     for (i = 1; i <= nCol; i++)
         {
          GetSQLDA(select_descriptor, i, SQLDA_DATABUF, &pData);
          if (SQLCODE == SQL_ERROR) error_handle();
          GetSQLDA(select_descriptor, i, SQLDA_DATABUF_TYPE, &dateType);
          if (SQLCODE == SQL_ERROR) error_handle();
          switch (dataType)
              {
               case SQL_C_CHAR:
                    printf(" %s ", pData);
                    break;
                case SQL_C_LONG:
                    printf(" %ld ",*(long *)pData);
                    break;
              }
         }
```

```
    }   /* end of while loop */

/* 10. EXEC SQL CLOSE cursor_name                                    */
EXEC SQL CLOSE demo_cursor;

/* 11. free user buffer                                              */
for (i = 1; i <= nHv; i++)
    {
     GetSQLDA(input_descriptor, i, SQLDA_DATABUF, &pData);
     if (SQLCODE == SQL_ERROR) error_handle();
     free(pData);
    }
for (i = 1; i <= nCol; i++)
    {
     GetSQLDA(select_descriptor, i, SQLDA_DATABUF, &pData);
     if (SQLCODE == SQL_ERROR) error_handle();
     free(pData);
    }
/* 12. De_allocate descriptor                                        */
free_descriptor_storage(input_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
free_descriptor_storage(select_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
```

➲ **Example 2**

Using SQLDA to retrieve multiple-row in a single FETCH statement:

```
#include "testdb.h"
/************************************************************/
 *   The example will show how to write a dynamic ESQL program using SQLDA
 *   and to query with an unknown number of input and output host variables
 *
 *   Table customer in database STORE is used in the following demo example.
 *   Schema of table customer is (cid int, lname(32), fname(32)).
 ************************************************************/
```

```
#define MAX_ENTRY   10
#define STRING_LEN  128
#define CONDITION   103
#define MAX_FETCH_ROWS   10
#define NUM_OF_FETCH_ROWS 5
/*****************************************************************
*  include header
*****************************************************************/
   EXEC SQL INCLUDE SQLCA;
   EXEC SQL INCLUDE SQLDA;
   EXEC SQL INCLUDE DBENVCA;

main()
{

   /*****************************************************************
         *  error handling
*****************************************************/
   EXEC SQL WHENEVER SQLERROR GOTO error_label;

   /*****************************************************************
         *  declare SQL host variables
*****************************************************/
   EXEC SQL BEGIN DECLARE SECTION;
   varchar cuser[8], passwd[8];
   varchar demoquery[64];
   varchar democursor[8];
   char dbname[18];                  /* char type is fix length string */
   int     nFetchRows = NUM_OF_FETCH_ROWS;
   EXEC SQL END DECLARE SECTION;

   /*****************************************************************
         *  declare variables
*****************************************************/
```

```
   int        i, rc = 0;
   char       fgConn = 0;
   long       datalen, colLen;
   long       nHv=0, nCol=0;
   char      *input_descriptor;
   char      *select_descriptor;
   char      *pData, *pColName;
   int       count;
      /*****************************************************************
         *  connect to database
*****************************************************/
   strcpy(dbname, TEST_DBNAME);       /* get db,user,password info */
   strcpy(cuser.arr, "SYSADM");
   cuser.len = strlen(cuser.arr);
   strcpy(passwd.arr, "");
   passwd.len = strlen(passwd.arr);
      EXEC SQL CONNECT TO :dbname :cuser :passwd;
   fgConn = 1;
   EXEC SQL SET AUTOCOMMIT OFF;
      /*****************************************************************
         *  step1. allocate SQLDA storage for input and output host variables
*****************************************************/
   rc = allocate_descriptor_storage(MAX_ENTRY, &input_descriptor);
   if (rc < 0) goto error_label;
   rc = allocate_descriptor_storage(MAX_ENTRY, &select_descriptor);
   if (rc < 0) goto error_label;
      /*****************************************************************
         *  clear all tuples of table customer
*****************************************************/
   EXEC SQL DELETE FROM customer;
   /*****************************************************************
         *  insert data for test
*****************************************************/
   EXEC SQL INSERT INTO customer VALUES(1000, 'aaa', 'test1');
```

```
   EXEC SQL INSERT INTO customer VALUES(2000, 'bbb', 'test2');
   EXEC SQL INSERT INTO customer VALUES(3000, 'ccc', 'test3');
   EXEC SQL INSERT INTO customer VALUES(4000, 'ddd', 'test4');
   EXEC SQL INSERT INTO customer VALUES(5000, 'eee', 'test5');
   EXEC SQL INSERT INTO customer VALUES(6000, 'fff', 'test6');
   EXEC SQL INSERT INTO customer VALUES(7000, 'ggg', 'test7');
   EXEC SQL INSERT INTO customer VALUES(8000, 'hhh', 'test8');
   EXEC SQL INSERT INTO customer VALUES(9000, 'iii', 'test9');
   EXEC SQL INSERT INTO customer VALUES(10000, 'jjj', 'test10');
   EXEC SQL INSERT INTO customer VALUES(11000);
   exec sql commit work;

   /*************************************************************
         *  specify the query demoquery with host variable, you can also ask
         *  user to input the query string from the terminal at run time
*****************************************************/
   sprintf(demoquery.arr, "%s %s",
                   "SELECT cid, lname, memo, memo2 FROM customer",
                   "WHERE cid > ?");
   demoquery.len = strlen(demoquery.arr);
      /*************************************************************
         *  step2. prepare the query
*****************************************************/
   EXEC SQL PREPARE demo_stmt FROM :demoquery;
      /*************************************************************
         *  step3. declare cousor for the query
*****************************************************/
   EXEC SQL DECLARE democursor SCROLL CURSOR FOR demo_stmt;
      /*************************************************************
         *  step4. describle input host variables information (including number,
         *         type, length, ...) and put them into SQLDA input_descriptor
         *         for reference
*****************************************************/
   EXEC SQL DESCRIBE BIND VARIABLES FOR demo_stmt INTO input_descriptor;
```

```
   rc = GetSQLDA(input_descriptor, 0, SQLDA_NUM_OF_HV, &nHv);
   if (rc < 0) goto error_label;
   printf("There are %d input host variables in the query \n\n", nHv);
      /*****************************************************************
         *  step5. set data type and buffer length of input buffer , and
         *         allocate it. Then, set these values.
         *         (note: assume the input data is character string less than
         *                128 bytes.)
*****************************************************/
   for (i = 1; i <= nHv; i++)
           {
           pData = MALLOC(STRING_LEN);
           rc = SetSQLDA(input_descriptor, i, SQLDA_DATABUF, pData);
           if (rc < 0) goto error_label;
           rc = SetSQLDA(input_descriptor, i, SQLDA_DATABUF_TYPE, SQL_C_CHAR);
           if (rc < 0) goto error_label;
           sprintf(pData, "%d", CONDITION);
           datalen = strlen(pData);
           rc = SetSQLDA(input_descriptor, i, SQLDA_INDICATOR, datalen);
           if (rc < 0) goto error_label;
           }
   /*****************************************************************
         *  step6. open cursor using data and information of
         *         SQLDA input_descriptor
*****************************************************/
   EXEC SQL OPEN democursor USING DESCRIPTOR input_descriptor;
   /*****************************************************************
         *  step7. describle output host variables information
         *         (including projection number, column name, column type,
         *         column length, ....) and put them into SQLDA select_descriptor
         *         for reference
*****************************************************/
   EXEC SQL DESCRIBE SELECT LIST FOR demo_stmt INTO select_descriptor;
```

```
   rc = GetSQLDA(select_descriptor, 0, SQLDA_NUM_OF_HV, &nCol);
   if (rc < 0) goto error_label;
   printf("There are %d columns returned \n\n", nCol);
   printColInfo(select_descriptor);
   /*******************************************************
         *  step8. bind output host variables buffer information (including buffer
         *         type, buffer length) and allocate buffers for each output host
         *         variables.
*******************************************************/
   bindBufInfo(select_descriptor);
   /*************************************************************
         *  step9. fetch data by cursor and print out the results, including:
         *         column name and data
*******************************************************/
   for (i = 1; i <= nCol; i++)
          {
          rc = GetSQLDA(select_descriptor, i, SQLDA_COLNAME, &pColName);
          if (rc < 0) goto error_label;
          printf("   %s    ", pColName);
          }
   printf("\n");
      for (i = 1; i <= nCol; i++)
          printf("========= ");
   printf("\n");

         do
                  {
                  EXEC SQL FETCH :nFetchRows ROWS democursor USING select_descriptor;
                  if (sqlca.sqlcode == SQL_NO_DATA_FOUND)
                           break;
#if 0
                  EXEC SQL GET BLOB column 2 FOR demo_stmt;
                  EXEC SQL GET BLOB column 3 FOR demo_stmt;
```

```
#endif
                /* print out result by buffer data type */
                printResult(select_descriptor);
                printf("\n");
                } while(sqlca.sqlcode == SQL_SUCCESS ||
                            sqlca.sqlcode == SQL_SUCCESS_WITH_INFO);
        printf("\n");
   /*************************************************************
        *  step10. close cursor
*****************************************************/
        EXEC SQL CLOSE democursor;
error_label:
   /*************************************************************
        *  print out error information
*****************************************************/
        if (sqlca.sqlcode)
           {
           printf("SQLSTATE: %ld \n", sqlca.sqlcode);
           printf("error code: %ld \n", sqlca.sqlerrd[0]);
           printf("error message: %s \n", sqlca.sqlerrmc);
           }
  /*******************************************************
        *  step11. deallocate SQLDA storage
*****************************************************/
        for (i = 1; i <= nHv; i++)
           {
           rc = GetSQLDA(input_descriptor, i, SQLDA_DATABUF, &pData);
           FREE(pData);
           }
        for (i = 1; i <= nCol; i++)
           {
           rc = GetSQLDA(select_descriptor, i, SQLDA_DATABUF, &pData);
           FREE(pData);
           }
```

```
        if (input_descriptor)
           {
           rc = free_descriptor_storage(input_descriptor);
           if (rc < 0)
                  {
                  printf("SQLSTATE: %ld \n", sqlca.sqlcode);
                  printf("error code: %ld \n", sqlca.sqlerrd[0]);
                  printf("error message: %s \n", sqlca.sqlerrmc);
                  }
           }
        if (select_descriptor)
           {
           rc = free_descriptor_storage(select_descriptor);
           if (rc < 0)
                  {
                  printf("SQLSTATE: %ld \n", sqlca.sqlcode);
                  printf("error code: %ld \n", sqlca.sqlerrd[0]);
                  printf("error message: %s \n", sqlca.sqlerrmc);
                  }
           }       /************************************************************
    *  disconnect from database
    ************************************************************/
        EXEC SQL WHENEVER SQLERROR CONTINUE;
        if (fgConn)
           EXEC SQL DISCONNECT;
}
/*****************************************************************
 *  printColInfo() --
 *    print out column information from SQLDA by describle SELECT LIST
 *****************************************************************/
void printColInfo(char *desc)
{
   int i, rc=0;
   long nCol=0;
```

```
char *pColName;
long   colType, colPrec, colScale, colNullable;
long   colLen, colNameLen;
rc = GetSQLDA(desc, 0, SQLDA_NUM_OF_HV, &nCol);

for (i = 1; i <= nCol; i++)
        {
        printf("column %ld information :\n", i);
        rc = GetSQLDA(desc, i, SQLDA_COLNAME_LEN, &colNameLen);
        rc = GetSQLDA(desc, i, SQLDA_COLNAME, &pColName);
        if (pColName != NULL)
                {
                printf("column name = %s \n", pColName);
                printf("column name length = %d \n", colNameLen);
                }

        rc = GetSQLDA(desc, i, SQLDA_COLTYPE, &colType);
        switch (colType)
                {
                case SQL_CHAR:
                   printf("column type = char \n");
                        break;
                case SQL_DECIMAL:
                        printf("column type = decimal \n");
                        break;
                case SQL_INTEGER:
                        printf("column type = integer \n");
                        break;
                case SQL_SMALLINT:
                        printf("column type = smallint \n");
                        break;
                case SQL_FLOAT:
                case SQL_REAL:
                        printf("column type = float \n");
```

```
                break;
        case SQL_DOUBLE:
                printf("column type = double \n");
                break;
        case SQL_VARCHAR:
                printf("column type = varchar \n");
                break;
        case SQL_DATE:
                printf("column type = date \n");
                break;
        case SQL_TIME:
                printf("column type = time \n");
                break;
        case SQL_TIMESTAMP:
                printf("column type = timestamp \n");
                break;
        case SQL_LONGVARCHAR:
                printf("column type = longvarchar \n");
                break;
        case SQL_BINARY:
                printf("column type = binary \n");
                break;
        case SQL_LONGVARBINARY:
                printf("column type = longvarbinary \n");
                break;
        case SQL_FILE:
                printf("column type = file \n");
                break;
        default:
                break;
        }

rc = GetSQLDA(desc, i, SQLDA_COLLEN, &colLen);
printf("column length = %ld \n", colLen);
```

```
          rc = GetSQLDA(desc, i, SQLDA_COLPREC, &colPrec);
          printf("column precision = %ld \n", colPrec);
          rc = GetSQLDA(desc, i, SQLDA_COLSCALE, &colScale);
          printf("column scale = %d \n", colScale);
          rc = GetSQLDA(desc, i, SQLDA_COLNULLABLE, &colNullable);
          if (colNullable == 0)
                 printf("column is not nullable \n");
          else
                 printf("column is nullable \n");
          printf("\n");
          }
}
/*************************************************************
 *  bindBufInfo() --
 *    set output host variables buffer information (including buffer type,
 *    buffer length) and allocate buffers for each output host variables.
 *************************************************************/
void bindBufInfo(char *desc)
{
   int  i, rc=0;
   char *pData;
   long   nCol=0, colType, dataType;
   long   dataLen;
   int  nFetch = MAX_FETCH_ROWS;
   rc = SetSQLDA(desc, 0, SQLDA_MAX_FETCH_ROWS, nFetch);
   rc = GetSQLDA(desc, 0, SQLDA_NUM_OF_HV, &nCol);
   for (i = 1; i <= nCol; i++)
          {
          rc = GetSQLDA(desc, i, SQLDA_COLTYPE, &colType);
          switch (colType)
                 {
                 case SQL_CHAR:
                 case SQL_VARCHAR:
                 case SQL_LONGVARCHAR:
```

```
                case SQL_BINARY:
                case SQL_LONGVARBINARY:
                case SQL_FILE:
                case SQL_DECIMAL:
                case SQL_DATE:
                case SQL_TIME:
                case SQL_TIMESTAMP:
                   pData = MALLOC(STRING_LEN*nFetch);
                        dataLen = STRING_LEN-1; /* for null terminate */
                        dataType = SQL_C_CHAR;
                        break;
                case SQL_INTEGER:
                        pData = MALLOC(4*nFetch);
                        dataLen = 4;
                        dataType = SQL_C_LONG;
                        break;
                case SQL_SMALLINT:
                        pData = MALLOC(2*nFetch);
                        dataLen = 2;
                        dataType = SQL_C_SHORT;
                        break;
                case SQL_FLOAT:
                case SQL_REAL:
                        pData = MALLOC(4*nFetch);
                        dataLen = 4;
                        dataType = SQL_C_FLOAT;
                        break;
                case SQL_DOUBLE:
                        pData = MALLOC(8*nFetch);
                        dataLen = 8;
                        dataType = SQL_C_DOUBLE;
                        break;
                default:
                        break;
```

```
                }
          rc = SetSQLDA(desc, i, SQLDA_DATABUF, pData);
          rc = SetSQLDA(desc, i, SQLDA_DATABUF_LEN, dataLen);
          rc = SetSQLDA(desc, i, SQLDA_DATABUF_TYPE, dataType);
          }
}
/*************************************************************
 *  printResult() --
 *    print out column data by their data type
*****************************************************/
void printResult(char *desc)
{
   int   i,j, rc=0;
   long   nCol=0, dataType[MAX_ENTRY+1];
   long   *ind[MAX_ENTRY+1];
   long   *pind;
   char *pData[MAX_ENTRY+1];
   int   retRows = pSQLCA->sqlerrd[3];
   int   dataLen[MAX_ENTRY+1];
   int   maxFetch;
   rc = GetSQLDA(desc, 0, SQLDA_NUM_OF_HV, &nCol);
   rc = GetSQLDA(desc, 0, SQLDA_MAX_FETCH_ROWS, &maxFetch);
   for (i = 1; i <= nCol; i++)
          {
      rc = GetSQLDA(desc, i, SQLDA_INDICATOR, &ind[i]);
          rc = GetSQLDA(desc, i, SQLDA_DATABUF_TYPE, &dataType[i]);
          rc = GetSQLDA(desc, i, SQLDA_DATABUF_LEN, &dataLen[i]);
          rc = GetSQLDA(desc, i, SQLDA_DATABUF, &pData[i]);
      }
   for (j = 0; j < retRows; j++)
      {
      for (i = 1; i <= nCol; i++)
         {
         if (*ind[i] == SQL_NULL_DATA)
```

```
            printf("NULL  ");
         else
            switch (dataType[i])
               {
               case SQL_C_CHAR:
                    printf(" %s ", pData[i]);
                    break;
               case SQL_C_LONG:
                    printf(" %l5d ", *(long *)pData[i]);
                    break;
               case SQL_C_SHORT:
                    printf(" %5d ", *(short *)pData[i]);
                    break;
               case SQL_C_FLOAT:
                    printf(" %f ", *(float *)pData[i]);
                    break;
               case SQL_C_DOUBLE:
                    printf(" %lf ", *(double *)pData[i]);
                    break;
               default:
                    break;
               }
         ind[i]++;
         pData[i] += dataLen[i];
         }
      printf("\n");
      }
}
```

➲ **To pass the values of the input host variables into SQLDA, set the following options**

- Allocate a valid data buffer for a host variable and set the input value into the data buffer according to the data type of the buffer.
- Set the pointer, type, length, and indicator of the data buffer.

NOTE: If you do not set the indicator value, DBMaker will use the length of the data buffer as the real input data length.

➲ **To get the values of the output host variables from SQLDA, set the following options**

- Allocate a valid data buffer for a host variable.
- Set the pointer, type, and length of the data buffer.

6.5 Dynamic ESQL BLOB Interface

Use the PUT BLOB or GET BLOB mechanisms in dynamic ESQL. In addition, Type 4 dynamic ESQL can use the same BLOB mechanism for static ESQL to *put* or *get* a BLOB.

In ESQL there are two BLOB styles —memory and file storage. BLOBs stored in memory are referred to as BLOB data and data stored in files are file objects. Details on how to PUT or GET BLOB data and file objects, with type 4 dynamic ESQL (SQLDA) are included in this section. In addition to the steps for the original type 4 dynamic ESQL, some extra steps must be considered for the BLOB interface.

Storing File Objects

You can store a file object by content or object name. Storing the file content in the database will be saved as data, and the external file will no longer be related to the database after storing it. Storing the file name in the database will be saved as data, and the content of the file object will still be stored in the external file.

Specify the type of the stored file object using SQLDA_STORE_FILE_TYPE in the SetSQLDA() function. Setting the option_value with the ESQL_STORE_FILE_CONTENT, means that the content of the specified file, with SQLDA_DATABUF, will be stored in the database. Setting the option_value, using ESQL_STORE_FILE_NAME means that the file name specified with SQLDA_DATABUF will be stored in the database. In *DBMaker,* the default value of SQLDA_STORE_FILE_TYPE is ESQL_STORE_FILE_CONTENT.

➲ **To store a file object, set the following options in SQLDA**

- Allocate a character data buffer for the file name, the maximum file name length is *MAX_FNAME_LEN = 79* and set the file name in the data buffer.
- Set the pointer of the data buffer *SQLDA_DATABUF* into *SQLDA*.
- Set the data buffer type *SQLDA_DATABUF_TYPE* with *SQL_C_FILE* into the data buffer type by *SQL_C_FILE* into *SQLDA*.

- Set file type SQLDA_STORE_FILE_TYPE with ESQL_STORE_FILE_CONTENT or ESQL_STORE_FILE_NAME into SQLDA.
- Set the indicator *SQLDA_INDICATOR* with real file name length into *SQLDA*.

➲ **Example**

To PUT data into the "**memo**" column from file object mary_memo.fo using the customer table **cid**, **cname**, **memo**:

```
#define maxNumber 10
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
char *input_descriptor;
char *pData;
long  datalen;
allocate_descriptor_storage(maxNumber, &input_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
strcpy(stmt_str.arr, "INSERT INTO customer VALUES(1, 'mary', ?)");
stmt_str.len = strlen(stmt_str.arr);
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
EXEC SQL DESCRIBE BIND VARIABLES FOR demo_stmt INTO input_descriptor;
pData = malloc(MAX_FNAME_LEN);
SetSQLDA(input_descriptor, 1, SQLDA_DATABUF, pData);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(input_descriptor, 1, SQLDA_DATABUF_TYPE, SQL_C_FILE);
if (SQLCODE == SQL_ERROR) error_handle();
/* Suppose we will put 'file name' into database */
SetSQLDA(input_descriptor. 1, SQLDA_STORE_FILE_TYPE, ESQL_STORE_FILE_NAME);
If (SQLCODE == SQL_SRROR) error_handing();)
/* Suppose mary's memo data is stored in file 'mary_memo.fo'            */
strcpy(pData, "mary_memo.fo");
datalen = strlen(pData);
```

```
SetSQLDA(input_descriptor, 1, SQLDA_INDICATOR, datalen);
if (SQLCODE == SQL_ERROR) error_handle();

EXEC SQL EXECUTE demo_stmt USING DESCRIPTOR input_descriptor;
/* suppose FreeSQLDA() can free SQLDA and all buffers allocated by application
*/
FreeSQLDA(input_descriptor);
```

Get a File Object

If you want to GET a file object, set the following options in SQLDA:

- Allocate a character data buffer for the file name (the maximum file name length is MAX_FNAME_LEN = 79) and set the file name into the data buffer (the file will store the data of the column "**memo**".
- Set the pointer of the data buffer into SQLDA by using the command SQLDA_DATABUF.
- Set the data buffer type SQLDA_DATABUF_TYPE with SQL_C_FILE into SQLDA.
- Set the data buffer maximum length SQLDA_DATABUF_LEN into SQLDA.

In this example, data will be retrieved from the column "**memo**" and placed into the file object (mary_memo.fo).

➲ **Example**

```
#define maxNumber 10
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
char *select_descriptor;
char *pData;
long  datalen;
allocate_descriptor_storage(maxNumber, &select_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
```

```
strcpy(stmt_str.arr, "SELECT memo FROM customer WHERE cname = 'mary'");
stmt_str.len = strlen(stmt_str.arr);
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
EXEC SQL DECLARE demo_cursor CURSOR FOR demo_stmt;
EXEC SQL OPEN demo_cursor;
EXEC SQL DESCRIBE SELECT LIST FOR demo_stmt INTO select_descriptor;
pData = malloc(MAX_FNAME_LEN);
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF, pData);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF_TYPE, SQL_C_FILE);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF_LEN, MAX_FNAME_LEN);
if (SQLCODE == SQL_ERROR) error_handle();
strcpy(pData, "mary_memo.fo");
EXEC SQL FETCH demo_cursor USING select_descriptor;
/* support PrintFile() can print out content of file                    */
printf("mary memo content - " \n);
PrintFile("mary_memo.fo");
EXEC SQL CLOSE demo_cursor;
/* support FreeSQLDA() can free SQLDA and all buffers allocated by application
*/
FreeSQLDA(select_descriptor);
```

Putting BLOB Data

If you want to PUT BLOB data using a type 4 dynamic ESQL, set the following options into SQLDA.

Before using the EXECUTE command:

- Allocate a data buffer for the input data.
- Set the pointer of the data buffer into *SQLDA* by using the *SQLDA_DATABUF* command.

- Set the data buffer type into *SQLDA* by using the command *SQLDA_DATABUF_TYPE*.
- Set the *BLOB* flag, *SQLDA_BLOB_FLAG* into *SQLDA* to specify that the host variable will *PUT* in the database.
- Before "*BEGIN PUT BLOB*" fill the data buffer with input data.
- Specify the length of this data *SQLDA_PUT_DATA_LEN* into *SQLDA*.

➲ **Example**

Data will be PUT into t*he column "*memo*"* from *the d*ata bufferData.

```
#define maxbufsize 256
#define maxNumber 10
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
char *input_descriptor;
char *pData;
long  datalen;
boolean fgEnd = FALSE;
allocate_descriptor_storage(maxNumber, &input_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
strcpy(stmt_str.arr, "INSERT INTO customer VALUES(1, 'mary', ?)");
stmt_str.len = strlen(stmt_str.arr);
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
EXEC SQL DESCRIBE BIND VARIABLES FOR demo_stmt INTO input_descriptor;
pData = malloc(maxbufsize);
SetSQLDA(input_descriptor, 1, SQLDA_DATABUF, pData);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(input_descriptor, 1, SQLDA_DATABUF_TYPE, SQL_C_CHAR);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(input_descriptor, 1, SQLDA_BLOB_FLAG, SQLDA_BLOB_ON);
if (SQLCODE == SQL_ERROR) error_handle();
EXEC SQL EXECUTE demo_stmt USING DESCRIPTOR input_descriptor;
```

```
EXEC SQL BEGIN PUT BLOB FOR demo_stmt;
/* support for mary's memo data is retrieved through function getInData(),  */
/* if no more input data will be retrieved, fgEnd will be assigned to TRUE  */
while(!fgEnd)
    {
     getInData(pData, maxbufsize, fgEnd);
     datalen = strlen(pData);
     SetSQLDA(input_descriptor, 1, SQLDA_PUT_DATA_LEN, datalen);
     if (SQLCODE == SQL_ERROR) error_handle();
     EXEC SQL PUT BLOB FOR demo_stmt;
    }

EXEC SQL END PUT BLOB FOR demo_stmt;
/* support FreeSQLDA() can free SQLDA and all buffers allocated by application */
FreeSQLDA(input_descriptor);
```

Get BLOB Data

If you want to retrieve BLOB data using type 4 dynamic ESQL, set the following options in SQLDA.

Before using Get BLOB:

- Allocate a data buffer for storing fetched data.
- Set the pointer of the data buffer *SQLDA_DATABUF* in *SQLDA*.
- Set the data buffer type *SQLDA_DATABUF_TYPE* in *SQLDA*.
- Set the data buffer maximum length by *SQLDA_DATABUF_LEN* into *SQLDA*.

Before FETCH:

- Allocate a data buffer for stored fetch data.
- Set the pointer of the data buffer in *SQLDA*.
- Set the data buffer type in *SQLDA*.

- Set the *BLOB* flag to specify that the column will get data in *SQLDA*.
- Set the maximum length of data buffer in *SQLDA*.
- Before *GET BLOB*: Specify *GET DATA* length to get data in *SQLDA*.

➲ **Example**

To GET data from the "**memo**" column in the data buffer "**pData**":

```
#define maxbufsize 256
#define maxNumber 10
EXEC SQL BEGIN DECLARE SECTION;
varchar stmt_str[128];
EXEC SQL END DECLARE SECTION;
char *select_descriptor;
char *pData;
long  datalen;
boolean fgEnd = FALSE;
allocate_descriptor_storage(maxNumber, &select_descriptor);
if (SQLCODE == SQL_ERROR) error_handle();
strcpy(stmt_str.arr, "SELECT memo FROM customer WHERE cname = 'mary');
stmt_str.len = strlen(stmt_str.arr);
EXEC SQL PREPARE demo_stmt FROM :stmt_str;
EXEC SQL DECLARE demo_cursor CURSOR FOR demo_stmt;
EXEC SQL OPEN demo_cursor;
EXEC SQL DESCRIBE SELECT LIST FOR demo_stmt INTO select_descriptor;
SetSQLDA(select_descriptor, 1, SQLDA_BLOB_FLAG, SQLDA_BLOB_ON);
if (SQLCODE == SQL_ERROR) error_handle();
EXEC SQL FETCH demo_cursor USING select_descriptor;
pData = malloc(maxbufsize);
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF, pData);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF_TYPE, SQL_C_CHAR);
if (SQLCODE == SQL_ERROR) error_handle();
SetSQLDA(select_descriptor, 1, SQLDA_DATABUF_LEN, maxbufsize);
if (SQLCODE == SQL_ERROR) error_handle();
```

```
printf("mary memo content - " \n);
do
        {
        EXEC SQL GET BLOB COLUMN 1 FOR demo_stmt;
If (SQLCODE ! = SQL_SUCCESS && SQLCODE != SQL_SUCCESS_WITH_INFO)
        Break;
        Printf(" %s ", pData);
        }
)
while (SQLCODE == SQL_SUCCESS || SQLCODE == SQL_SUCCESS_WITH_INFO)
    {
     EXEC SQL GET BLOB COLUMN 1 FOR demo_stmt;
     printf(" %s ", pData);
    }
EXEC SQL CLOSE demo_cursor;
/* support FreeSQLDA() can free SQLDA and all buffers allocated by application */
FreeSQLDA(select_descriptor);
```

➲ **To GET the total data length for a column before using GET BLOB to GET data from a column**

- Set SQLDA_DATABUF_LEN with 0.
- *GET BLOB* from the column.
- Get value of *SQLDA_INDICATOR,* the value is total data length of the column. To GET the total data length for a column before using GET BLOB to GET data from a column.

7 Project and Module Management

When using the *DBMaker* ESQL/C preprocessor *dmppcc* to preprocess an ESQL/C source file, provide the database name, user name, and password. The user name you use for preprocessing the ESQL/C source file must have connect and resource privileges for connecting to the database and preprocessing the file.

dbname, dbuser and dbusr_passwd can be an identifier or the char type's host variable if it has been declared in the ESQL/C applications declare section. dbusr_passwd can be ignored, if the user has no password. It is not necessary to use the same database user name, dbuser, for preprocessing the ESQL/C program and execution time in the ESQL/C program. For example, a banking database with the user "**acc_dba**" for developing the "**bank**" program.

➲ **Syntax**

```
dmppcc -d test_db -u db_user_id -p db_user_passwd esql_source.ec
```

➲ **Example 1**

When writing an ESQL/C source file, you should add a CONNECT statement in the source file to enable the application program to connect to the database.

```
EXEC SQL CONNECT TO dbname dbuser dbusr_passwd;
```

PARAMETER	SYNTAX
Dbname	[identifier | :host_variable_identifier]
Dbuser	[identifier | :host_variable_identifier]
dbusr_passwd	[| identifier | :host_variable_identifier]

Table 7-1 Connect Statement Parameters and Syntax

➲ **Example 2**

In the shell command, type User acc_dba from the bank database.

```
dmppcc -d bank -u acc_dba -p acc_dba connect.ec
```

➲ **Example 3**

Contents of the file connect.ec:

```
EXEC SQL INCLUDE DBENVCA;
EXEC SQL INCLUDE SQLCA;
connect()
    {
     /* use the clerk account to connect to the bank database          */
     EXEC SQL CONNECT TO bank clerk pd_clerk;
     if (SQLCODE) return SQLCODE;
     ...
     do_clerk_operation();
     ....
    }
```

7.1 Project and Module objects

When preprocessing an ESQL/C source file, *DBMaker* will create an execution plan and store all related information in the database in a component called a module. If you do not specify the module name, the default module name will be the ESQL/C source file name.

An error occurs when user tries to access an older version of an execution plan. When other ESQL developer's preprocess the same ESQL program again, dmppcc will delete the previously stored plan, and then create the new stored plan. If you often find the error message "the executable may be out of date, please rebuild it" when you execute the ESQL program, you should compile the related .c file and re-link to executables.

However, although this is an executable version error, you may still want to ignore it where there are many developers developing different ESQL modules in the same ESQL application, use the "-n" option to ignore this error message when you are in the developing phase. To optimize performance and reduce problems in ESQL project management, you should remove the "-n" option after you finish coding your application.

Any developer's application system may contain more than one ESQL/C module. If the developer tries to manage (grant/revoke or drop privilege) every module individually, the burden will be big. The purpose of a project is to let the developer group ESQL/C modules all together, and organize the application system more easily. After preprocessing an ESQL/C source file, *dmppcc* will store the project name in addition to the module name. If you do not specify the project name, the default project name will be the same as the module name.

When preprocessing any ESQL source file, if the project does not exist in the database, *DBMaker* will automatically create a project to store the module. If the project already exists, *DBMaker* will also automatically associate the new module to that project. Each module can only be associated with one project.

To look for the information on ESQL projects and modules, you can reference the database system table SYSPROJECT by an *SQL* statement.

➲ **Example**

```
select * from SYSPROJECT;
```

COLUMN	MEANING
PROJECT_NAME	ESQL project name
PROJECT_OWNER	ESQL project owner
MODULE_NAME	ESQL module name
MODULE_OWNER	ESQL module owner
MODULE_SOURCE	Module's source file name
REF_CMD	Referenced command number (used internally by dmppcc)

Table 7-2 SYSPROJECT System Table

The information of the ESQL execution plan is stored in the SYSCMDINFO system table. This system table not only stores the ESQL execution plan, but also other execution plans for stored commands and procedures. You may reference the manual on stored commands for more detailed information about each field.

If you set option –cs or –n in dmppcc, DBMaker will not store the execution plan, module, project, or owner name of related table. To prevent an error caused by different ESQL/C preprocessor users and the program execution user, it is recommended that you put the owner name for every table in the SQL statement when you set –cs or –n option.

COLUMN	MEANING
MODULENAME	ESQL module name
CMDNAME	ESQL preprocessor generated command name
CMDOWNER	ESQL preprocessor generated command owner
STATEMENT	Original SQL statement
DATA	Execution plan data
DESCPARM	Description parameters
STATUS	Valid or invalid

Table 7-3 SYSCMDINFO System Table

Dropping a Project

Since projects are used for maintaining the relationship of the ESQL modules, when the project is no longer useful, you can use the DROP PROJECT statement to drop all the related execution plans and information for the project.

You can also remove a module from a project and all stored commands related to the module from the database. When a project contains only one module, the project will also be removed from the database.

Only the project owner or database administrator can drop a project or module and grant or revoke execution privileges to other users.

➲ **Example**

DROP PROJECT syntax:

```
DROP PROJECT project_name;
```

Loading or Unloading Projects or Modules

You can use the LOAD or UNLOAD PROJECT or MODULE functions in dm*SQL* to unload or load the related project or any specified module. For more information see the UNLOAD/LOAD syntax in the *dmSQL User's Guide*.

➲ **Example 1**

The UNLOAD syntax:

```
UNLOAD PROJECT FROM [ower_pattern.]project_pattern TO script_name
UNLOAD MODULE [ower_pattern.]module_pattern FROM PROJECT [owner_name.]project_name
TO script_name.
```

➲ **Example 2**

To use UNLOAD:

```
dmSQL> UNLOAD PROJECT FROM project1 TO project.scr;
dmSQL> UNLOAD MODULE module1 FROM PROJECT project1 TO module.scr;
```

➲ **Example 3**

The LOAD syntax:

```
LOAD PROJECT FROM script_name.
LOAD MODULE FROM script_name.
```

➲ **Example 4**

To use the LOAD command:

```
dmSQL> LOAD PROJECT FROM project.scr;
dmSQL> LOAD MODULE FROM module.scr;
```

NOTE: *The UNLOAD/LOAD function can only be used in dmSQL.*

Granting or Revoking Privileges for Projects

Execution privileges can be granted or revoked to other users for a project. When any user tries to execute a project for which they don't have authority to execute, it will return an error at run time. Because the project is for the developer to group or manage modules in the database, it's possible that an application system has many linked to modules from different projects. In this case, the application user can only execute the part for which they have the execution privilege.

➲ **Syntax**

GRANT or REVOKE *SQL* syntax:

```
GRANT EXECUTE ON PROJECT project_name TO auth_user_list;
REVOKE EXECUTE ON PROJECT project_name FROM auth_user_list;
```

The authority information related to ESQL is stored in the SYSAUTHEXE system table. You may look in the table to check for authorized users.

COLUMN	MEANING
OBJNAME	Project name
OWNER	Project owner
OBJTYPE	PROJECT or STORE COMMAND or STORE PROCEDURE
GRANTEE	Authorized user

Table 7-4 SYSAUTHEXE System Table