DBMaker

ODBC 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

©Copyright 1995-2003 by CASEMaker Inc.
Document No. 645049-231123/DBM41-M07212003-ODBC

Publication Date: 2003-07-21

All rights reserved. No part of this manual may be reproduced, stored in a retrieval system, or transmitted in any form, without the prior written permission of the manufacturer.

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 1-1

1.1 Additional Resources 1-3

1.2 Technical Support 1-4

1.3 Document Conventions 1-5

2 Example Application 2-1

2.1 Library Model 2-2

2.2 Required Files 2-4

Header Files 2-4

Link Libraries 2-4

2.3 Example ODBC Application 2-6

2.4 Compiling and Linking 2-8

2.5 Sample Programs 2-9

3 Database Connections 3-1

3.1 Environment Handle 3-4

3.2 Connection Handle 3-5

3.3 Connecting to a Data Source 3-6

SQLConnect 3-6

SQLDriverConnect 3-10

©Copyright 1995-2003 CASEMaker Inc.

Multiple Connections 3-13
3.4 Connect Options 3-15
SQLSetConnectOption 3-15
SQLGetConnectOption 3-16
3.5 Freeing Handles 3-17
SQLDisconnect 3-17
SQLFreeConnect 3-17
SQLFreeEnv 3-18
4 SQL Statements 4-1
4.1 The SQL Language 4-3
The Role of SQL with ODBC 4-3
Basic SQL Statements 4-4
Data Definition Language (DDL) 4-4
Data Manipulation Language (DML) 4-5
4.2 Executing SQL Statements 4-11
SQLAllocStmt 4-11
SQLExecDirect 4-12
SQLRowCount 4-13
SQLFreeStmt 4-14
SQLPrepare and SQLExecute 4-15
4.3 Parameters 4-17
Parameter Functions 4-17
Using Parameters in SQLExecDirect 4-26
Clearing Bound Parameters 4-27
4.4 Entering Large Data 4-29
How to Enter Large Data 4-29
Canceling the Execution of SQLPutData 4-34
Placing Large Data in a File Object 4-34
4.5 Get and Set Options 4-36
5 Retrieving Results 5-1
5.1 Queries Using ODBC 5-3

©Copyright 1995-2003 CASEMaker Inc.

Binding Storage Locations and Fetching Data 5-3
Result Columns Characteristics..... 5-6
More about Result Columns..... 5-11
Clear Bound Columns 5-14

5.2 Cursors 5-15

When to Use Cursors..... 5-15
Getting the Cursor Name 5-15
Using Cursors 5-16
Setting the Cursor Name 5-17

5.3 Fetching Large Data 5-19

SQLGetData..... 5-21
Stopping SQLGetData Operations 5-25
Binding Columns to Retrieve File Objects..... 5-25
Fetching the Filename of File Objects..... 5-26

5.4 Manipulating Result Sets 5-27

Rowsets..... 5-27
Program Flow 5-27
Storage Binding 5-28
Positioning the Cursor 5-34
Arguments of SQLExtendedFetch 5-34
Returning Values and Processing Errors..... 5-39
Table Modification Using SQLSetPos 5-42
Column Indicators..... 5-48
SQLPutData..... 5-49
Using SQLSetPos..... 5-53

6 Error Handling.....6-1

6.1 Retrieving Error Information 6-2

Common Error Codes Defined in ODBC..... 6-2
How to Use SQLError..... 6-2
Error Queues 6-5

6.2 Catalog Functions.....6-7

Search Patterns..... 6-7

©Copyright 1995-2003 CASEMaker Inc.

SQLTables 6-8
SQLColumns 6-11
SQLStatistics 6-13
SQLSpecialColumns 6-14

6.3 System Information 6-17

SQLGetTypeInfo 6-17
SQLGetInfo 6-20
SQLGetFunctions 6-21

6.4 Procedure Information 6-23

SQLProcedureColumns 6-23
SQLProcedures 6-26

7 Transaction Control 7-1

7.1 Transactions and Savepoints 7-2

7.2 Terminating a Transaction 7-5

7.3 Auto-Commit & Manual-Commit 7-7

8 ODBC 3.0 Functions 8-1

8.1 Deprecated functions 8-2

8.2 Modified functions 8-4

SQLCancel 8-4
SQLColumns 8-4
SQLFetch 8-5
SQLGetData 8-5
SQLGetFunctions 8-5
SQLGetInfo 8-6
SQLProcedureColumns 8-6

8.3 New functions 8-8

SQLAllocHandle 8-8
SQLBulkOperations 8-9
SQLCloseCursor 8-10
SQLColAttribute 8-11
SQLCopyDesc 8-14

©Copyright 1995-2003 CASEMaker Inc.

SQLEndTran 8-16
SQLFetchScroll 8-16
SQLForeignKeys 8-18
SQLFreeHandle 8-20
SQLGetConnectAttr 8-21
SQLGetDescField 8-22
SQLGetDescRec 8-25
SQLGetDiagField 8-26
SQLGetDiagRec 8-28
SQLGetEnvAttr 8-29
SQLGetStmtAttr 8-29
SQLPrimaryKeys 8-31
SQLSetConnectAttr 8-32
SQLSetDescField 8-33
SQLSetDescRec 8-36
SQLSetEnvAttr 8-38
SQLSetStmtAttr 8-39

9 Unicode Support 9-1

9.1 Unicode Encoding Interfaces 9-2

A Function Sequence Differences 1

A.1 SQLRowCount 2

A.2 SQLGetCursorName 3

B Function Property Differences 1

B.1 SQLPutData 2

B.2 SQLColumns 3

B.3 SQLTables 4

B.4 SQLDriverConnect 5

B.5 SQLBindParameter 6

B.6 Positioned DELETE/UPDATE 7

B.7 SQLSetConnectOption 8

©Copyright 1995-2003 CASEMaker Inc.

B.8 SQLGetConnectOption 11
C ODBC 3.0 Errors 1
C.1 SQLParamData 2
C.2 SQLPrepare 3
D Data Types 1
D.1 ODBC SQL Data Types 2
D.2 ODBC C Data Types 4
D.3 Default ODBC C Data Types 6
D.4 Precision, Scale, Length, and Display Size 7
D.5 Data Type Conversions 9
SQL to C Data Conversion 9
C to SQL Data Conversion 14
E ODBC Log Function 1

©Copyright 1995-2003 CASEMaker Inc.

1 Introduction

Welcome to the ODBC Programmer'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 query your database using existing ODBC-compliant applications.

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

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

This guide is intended for programmers who want to create front-end applications for DBMaker. Users should be familiar with the C programming language (hereafter

©Copyright 1995-2003 CASEMaker Inc.

referred to as C), and should have a C development tool available if they wish to compile and execute the example programs.

Information on C programming is beyond the scope of this book, and users should consult a C programming guide if they encounter any problems in this area. If you encounter any problems with compiling and running the example programs with your development tool, you should consult your development tool documentation or the development tool vendor.

This guide introduces the DBMaker ODBC API and outlines how to construct a front-end application for a database using the DBMaker ODBC API. Since this book is only intended as an introduction to ODBC programming, not all ODBC concepts and practices may be covered fully. However, all concepts that are presented will be covered in enough depth to let you understand what is happening in the example programs, and why.

Each chapter introduces a group of related functions and their options, and explains any differences you may encounter between the DBMaker ODBC API and the Microsoft ODBC 2.1 specification (For information about DBMaker*'s* ODBC 3.0 API, please refer to chapter 8 on ODBC 3.0 Functions. You will learn how to use each function, and how each function fits into a program as a whole.

Examples and illustrations are provided whenever possible to help you understand the information presented. Example programs are given using C and can be compiled with any suitable C/C++ compiler.

Although this guide provides information on all DBMaker ODBC functions, it is not intended as a comprehensive reference to the Microsoft ODBC 3.0 API. When using this guide, you may find it helpful to have a reference work available that has detailed information on all functions and state transitions. The recommended reference is "ODBC 3.0 Programmer's Reference" by Microsoft Press.

©Copyright 1995-2003 CASEMaker Inc.

1.1 Additional Resources

DBMaker provides a complete set of DBMS manuals in addition to this one. For more detailed 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 dmSQL interface tool, refer to the "*dmSQL User's Guide*".
- For more information on the SQL language used in dmSQL, refer to the "*SQL Command and Function Reference*".
- For more information on the ESQL/C programming, refer to the "*ESQL/C User's Guide*".
- For more information on error and warning messages, refer to the "*Error and Message Reference*".
- For more information on the DCI COBOL Interface, refer to the "*DCI User's Guide*".

©Copyright 1995-2003 CASEMaker Inc.

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

©Copyright 1995-2003 CASEMaker Inc.

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. Other forms of this convention include Prototype and Syntax.
`CommandLine`	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

Table1 Document Conventions Table

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.

2 Example Application

After installing the ODBC software and DBMaker's ODBC driver, begin constructing a simple ODBC application using DBMaker. In this chapter, we will teach you how to compile and link an ODBC program with DBMaker's ODBC driver.

©Copyright 1995-2003 CASEMaker Inc.

2.1 Library Model

Create an ODBC application using the DBMaker function library or the ODBC function library. If the application program needs to make low-level system calls to the operating system, file system, or hardware-specific drivers, you may also need to use your own function libraries. *Figure 2-1* and *Figure 2-2* show the models for interfacing with these libraries.

Figure 2-1 models the use of the ODBC Driver Manager, and *Figure 2-2* models the use of the DBMaker Driver directly without using the ODBC Driver Manager. When using the *ODBC Driver Manager* you may connect to data sources other than DBMaker. Without *Driver Manager*, your application links to the DBMaker function library directly and achieves better performance, but loses the ability to connect to other data sources.

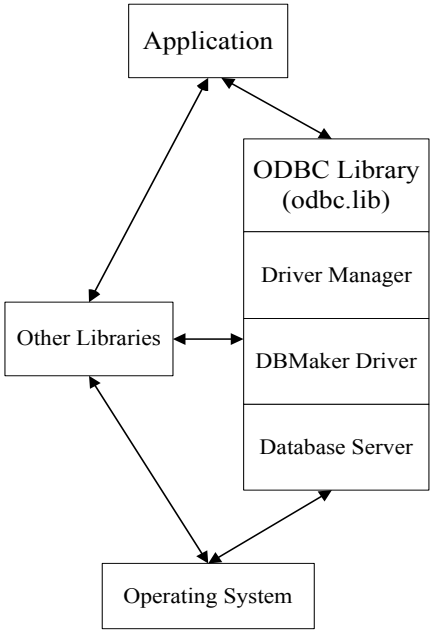

Table2: ODBC library model when using the ODBC Driver Manager

©Copyright 1995-2003 CASEMaker Inc.

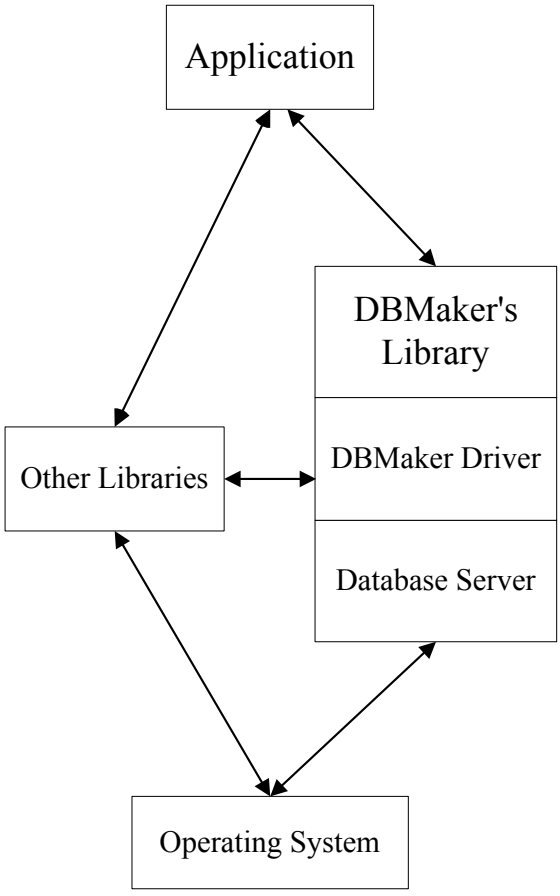

Table3: ODBC library model when using the DBMaker *driver directly*

©Copyright 1995-2003 CASEMaker Inc.

2.2 Required Files

To construct an ODBC program using the DBMaker ODBC driver, specify the header files and linking libraries in the makefile. In the following description, we use C with DBMaker version 3.5 as an example.

Header Files

When creating an ODBC application, the following header files are required: ***SQL.h***, ***SQLext.h***, ***SQLopt.h*** and ***SQLunix.h*** (***SQLunix.h*** is required for UNIX only).

SQL.h and ***SQLext.h*** are the standard ODBC include files, and DBMaker *provides* ***SQLopt.h*** and ***SQLunix.h*** for some driver-specific options and for UNIX applications, respectively. Since ***SQLext.h*** also includes ***SQL.h***, you only need to include ***SQLext.h*** in the program.

The header files are the same on all platforms, except ***SQLunix.h***, which is used only for programs running on the UNIX platform.

In Microsoft Windows, these files can be found in the ***c:\dbmaker35\include\c*** directory, assuming that DBMaker was installed in the default installation directory. Otherwise, they can be found under ***d:\install_directory\include***, where ***d:*** is the drive DBMaker was installed on, and ***install_directory*** is the directory.

In a UNIX environment, these files are located under the ***~dbmaker/3.5/include*** directory.

Link Libraries

Link libraries are also required when creating ODBC applications for use with DBMaker. Which link libraries will be used depends on the platform the application will be running on.

©Copyright 1995-2003 CASEMaker Inc.

WINDOWS 95/98/NT/2000

For Windows ODBC applications:

If using the Driver Manager, link the library ***odbc.lib*** in the ODBC SDK, often located in ***c:\odbcsdk\lib\odbc.lib***. You must register DBMaker in ***odbc.ini*** first so that the Driver Manager can load the DBMaker driver correctly. In this case, you do not have to specify ***dmapi35.lib*** in the makefile. The Driver Manager will load the necessary DLL automatically.

NOTE *If you are not using Driver Manager, link the library* **dmapi35.lib** *provided by DBMaker. This library is normally found in* **c:\dbmaker35\lib**.

There is one dynamic link library in the ***c:\windows\system*** directory that is used with ODBC programs, ***dmapi35.dll***. However, programmers only need to link the libraries ***odbc.lib*** (with ***Driver Manager***) or ***dmapi35.lib*** (without ***Driver Manager***). After these libraries are linked, the application will call this DLL automatically.

UNIX

On UNIX platforms, you must link the ***libdmapic.a*** file when creating a client/server ODBC application.

©Copyright 1995-2003 CASEMaker Inc.

2.3 Example ODBC Application

Consider the following example program in the UNIX environment.

➲ **Example**

To connect to a data source and then use SQLGetInfo to retrieve the DBMS version:

```
#include <stdio.h>
#include "sqlext.h"
#include "sqlopt.h"
#include "sqlunix.h"

#define  STR_LEN 30
HENV     henv;                        /* environment handle         */
HDBC     hdbc;                        /* connection handle          */
HSTMT    hstmt;                       /* statement handle           */
SDWORD   retcode;                     /* return code                */
UCHAR    info[STR LEN];               /* info string for SQLGetInfo */

retcode = SQLAllocEnv(&henv);
retcode = SQLAllocConnect(&hdbc);
retcode = SQLConnect(hdbc, "TEST", SQL NTS, "SYSADM", SQL NTS, "",
                     SQL_NTS);

if (rc != SQL_SUCCESS)
    goto EXIT;
rc = SQLGetInfo(hdbc,SQL_DBMS_VER, &info, STR_LEN,&cbInfoValue);

if (rc != SQL_SUCCESS)
    goto EXIT;
printf("Current DBMS version is %s\n", info);
EXIT:
```


©Copyright 1995-2003 CASEMaker Inc.

```
    SQLDisconnect(hdbc);

    SQLFreeConnect(hdbc);

    SQLFreeEnv(henv);

    return;
```

©Copyright 1995-2003 CASEMaker Inc.

2.4 Compiling and Linking

The example uses the compiler ***acc*** and ***$dir*** is the DBMaker directory.

➲ **Example 1**

To compile the example program (*example.c*) type the following on the UNIX command line:

```
sh> acc -c example.c -I$dir/dbmaker/include
```

Now link the example program (***example.o***) with DBMaker's library ***libdmapis.a.***

➲ **Example 2**

To produce an executable file called example:

```
sh> acc -o example example.o -L$dir/dbmaker/lib -ldmapis
```


©Copyright 1995-2003 CASEMaker Inc.

2.5 Sample Programs

Additional ODBC sample programs are provided with DBMaker in the ***samples*** directory. You can use the makefile in this directory to build and execute them. First change to the ***dbmaker/samples*** directory, then type "make ex1" to build the executable ***ex1*** for the sample program ***ex1.c***.

There are several different C language development tools in Windows, such as Microsoft Visual C++, Borland C++ and Watcom C++. You may need to edit your makefile in order to set the directories to include files and linking libraries in different ways for different development tools. For example, in Visual C++ you have to open a new project and edit the ***.mak*** and ***.def*** file.

DBMaker provides an example makefile for Visual C++ in the ***c:/dbmaker/samples/c*** directory.

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.

3 Database Connections

In any ODBC application, you must properly setup the ODBC environment and connect to a data source before executing SQL statements or performing queries. Similarly, you must disconnect from the database and free the memory allocated for the ODBC environment when the program terminates. This chapter introduces the functions necessary to setup a connection to a data source.

In this chapter you will learn how to:

- Initialize the *ODBC* environment by allocating environment and connection handles using the *SQLAllocEnv* and *SQLAllocConnect* functions.
- Establish a connection to a predetermined data source using the *SQLConnect* function or to an unknown data source using the *SQLDriverConnect* functions.
- Use the connection *SQLGetConnectOption* and *SQLSetConnectOption* options to connect to a data source.
- Disconnect from a data source using the *SQLDisconnect* function.
- Free the environment and connection handles when the application terminates using the *SQLFreeConnect* and *SQLFreeEnv* functions.

NOTE *You allocate environment and connections handles, and get and set connection options differently using DBMaker (ODBC 3.0) than what is described in this chapter. For more information, refer to chapter 8 on "ODBC 3.0 Functions".*

©Copyright 1995-2003 CASEMaker Inc.

The following program flow chart is for an application that uses all of the six ODBC functions.

©Copyright 1995-2003 CASEMaker Inc.

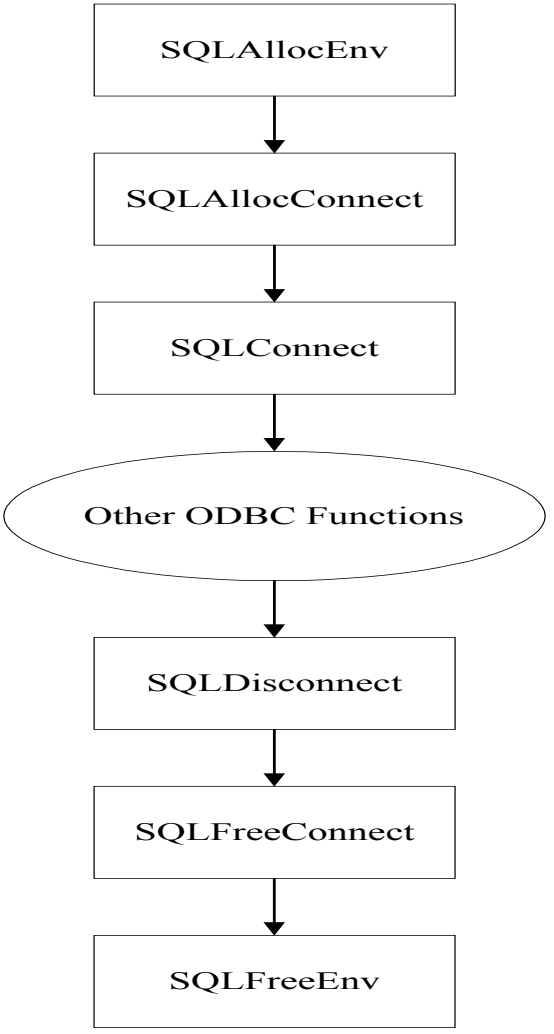

Table4: Program flowchart for connecting to and disconnecting from a data source.

©Copyright 1995-2003 CASEMaker Inc.

3.1 Environment Handle

In an ODBC application, the SQLAllocEnv function is called to setup the ODBC environment before calling any other ODBC functions. When you call the SQLAllocEnv function, the DBMaker driver allocates an area of memory for environment information and returns an environment handle to the application.

The environment handle identifies the area of memory the DBMaker driver will use to store global information about the ODBC environment. This may include information such as a list of valid connection handles and the currently active connection handle. Only one environment handle should be allocated in the application.

Prototype

SQLAllocEnv:

```
RETCODE SQLAllocEnv(HENV FAR * phenv)
```

Example 1

To allocate an environment handle, declare a variable of type HENV:

```
HENV henv1;
```

Example 2

To call SQLAllocEnv and pass the address of the HENV variable:

```
retcode = SQLAllocEnv(&henv1);
```

The environment handle is now a valid handle, and you can use it later in the application. If the application calls SQLAllocEnv with a pointer to a valid environment handle, the driver will overwrite the previous contents of the environment handle.

©Copyright 1995-2003 CASEMaker Inc.

3.2 Connection Handle

After allocating an environment handle, a connection handle is allocated before connecting to any ODBC data source. A connection handle identifies memory storage for each connection in an ODBC application, and contains information such as the database name and username. The SQLAllocConnect function allocates memory for a connection handle.

➲ **Prototype**

SQLAllocConnect:

```
RETCODE SQLAllocConnect(HENV henv, HDBC * phdbc)
```

➲ **Example 1**

To allocate a connection handle, declare a variable of type HDBC:

```
HDBC hdbc1;
```

➲ **Example 2**

To call SQLAllocConnect and pass the address of the variable:

```
retcode = SQLAllocConnect(henv1, &hdbc1);
```

©Copyright 1995-2003 CASEMaker Inc.

3.3 Connecting to a Data Source

Connecting to a data source before attempting to access the data contained within is required. To connect to a data source, the data source with a valid connection handle (**hdbc**) must first be specified. A data source connection can be performed with or without a dmconfig.ini file residing on the client site.

SQLConnect

Use SQLConnect to establish a connection between a data source and a valid connection handle.

➲ **Prototype**

SQLConnect:

```
RETCODE SQLConnect(
                    HDBC          hdbc,
                    UCHAR  FAR *  szDSN,
                    SWORD         cbDSN,
                    UCHAR  FAR *  szUID,
                    SWORD         cbUID,
                    UCHAR  FAR *  szAuthStr,
                    SWORD         cbAuthStr);
```

An application has to pass the following to use SQLConnect:

- A valid connection handle, not currently connected to another data source.
- The name of the data source and length of the name.
- User identification and its length.
- A password and its length.

©Copyright 1995-2003 CASEMaker Inc.

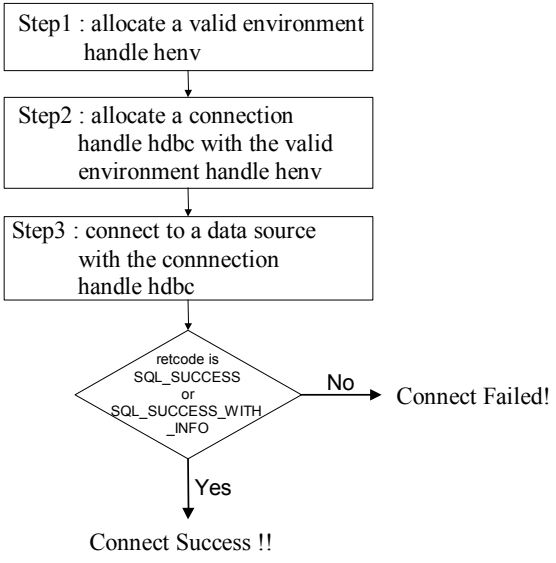

Table5: Program flow for connecting to a data source

1. In step one, **SQLAllocEnv** is called to allocate an environment handle **henv**.
2. In step two, **SQLAllocConnect** is called to allocate a valid connection handle **hdbc** with a valid environment handle **henv**.
3. In step three, **SQLConnect** is called to connect to a data source with a valid connection handle (**hdbc**).
4. If the code returned is; **SQL_SUCCESS** or **SQL_SUCCESS_WITH_INFO**, the connection has been established correctly.

➲ **Example 1**

To connect to data source *TEST_DB* with user *MYNAME* and password *PASS*:

```
retcode = SQLConnect(hdbc, "TEST_DB", SQL_NTS,
                           "MYNAME",  SQL_NTS,
                           "PASS",    SQL_NTS);
```

©Copyright 1995-2003 CASEMaker Inc.

SQL_NTS means the string is **NULL** terminated, and the driver calculates the length of the string. When using SQLConnect the data source name is a required argument, and the user identification and password are optional.

With DBMaker, a default username and password can be placed in the ***dmconfig.ini*** file to omit the use of them in the SQLConnect connection string. The driver then acquires the username and password specified in ***dmconfig.ini***.

➲ **Example 2**

To set DB_USRID=MYNAME and DB_PASWD=PASS in *dmconfig.ini* and then call TEST_DB:

```
retcode = SQLConnect(hdbc, "TEST_DB", SQL_NTS, "", SQL_NTS,
                           "", SQL_NTS);
```

When an application calls **SQLConnect**, the ***Driver Manager*** uses the data source name (**test_db**) to read the name of the driver DLL from the appropriate section of the ***ODBC.INI*** file. Then it loads the driver DLL and passes the username and password arguments to the driver. No dmconfig.ini is required for the client site.

USING SQLCONNECT WITHOUT A DMCONFIG.INI FILE

DBMaker will support connecting to a database without a dmconfig.ini file being required for the client site. Specifying the settings for the dmconfig.ini can be performed using the SQLConnect() connection string and suitable keyword(s). A second argument of SQLConnect() will have the capability to accept a special connection string.

DBMaker will use the default values for each keyword if no keyword(s) are specified in the SQLConnect() connection string and no dmconfig.ini is present for the client site.

The following keywords can be used:

- DSN: data source name
- DIFCO: merge connection or not (please refer DB_DIFCO definition of dmconfig.ini)

©Copyright 1995-2003 CASEMaker Inc.

- CTIMO: connection time out (please refer DB_CTIMO definition of dmconfig.ini)
- ATCMT: autocommit on or off (please refer DB_ATCMT definition of dmconfig.ini)
- STRSZ: the length of returned data of the STRING type, used only by UDF (please refer DB_DIFCO definition of dmconfig.ini)
- STROP: the keyword specifies whether space-padding needs to be removed before applying the string concatenation operator (||) (please refer DB_DIFCO definition of dmconfig.ini)
- SVADR: remote data source address (please refer DB_DIFCO definition of dmconfig.ini)
- PTNUM: port number (please refer DB_DIFCO definition of dmconfig.ini)

The keyword 'DSN' pair must be placed in the first position of the SQLConnect string; 'SVADR' and 'PTNUM' must be placed in the second and third position. The other keyword pairs have no special positioning rule.

In the prototype for SQLConnect mentioned above, the original meaning of szDSN is "data source name", but now it is changed to 'connection string'. The format of for the input string is "keyword1=value1; keyword2=value2;, …".

➲ **Example 1**

```
SQLConnect(hdbc, "DSN=TEST_DB;SVADR=172.0.0.1;PTNUM=12345;",
                    SQL_NTS, ....);
```

➲ **Example 2**

```
SQLConnect(hdbc, "DSN=DBSAMPLE;SVADR=172.0.0.1;PTNUM=12345;ATCMT=0;
                    CTIMO=2;", SQL_NTS, ....);
```

©Copyright 1995-2003 CASEMaker Inc.

SQLDriverConnect

Use SQLDriverConnect to connect to a specific predetermined data source. Driver Manager can be used to display all of the available data sources to provide the user with a connection list for the data sources.

Prototype

SQLDriverConnect:

```
RETCODE SQLDriverConnect(
                        HDBC        hdbc,
                        HWND        hwnd,
                        UCHAR  FAR *szConnStrIn,
                        SWORD       cbConnStrIn,
                        UCHAR  FAR *szConnStrOut,
                        SWORD       cbConnStrOutMax,
                        SWORD  FAR *pcbConnStrOut,
                        UWORD       fDriverCompletion);
```

An application has to pass the following information to SQLDriverConnect:

- A valid connection handle that is not yet associated with a data source.
- A valid window handle to provide a parent window for the dialog box.
- The input connection string (*szConnStrIn*) and its length. The connection string has its own special syntax (see the section on connection strings) and contains specific information needed to connect to a data source. If the input information is not complete, *SQLDriverConnect* will use a dialog box to request more information from the user before it sends connection information to the database driver.
- The output connection string (*szConnStrOut*) and its length. This is the final piece of connection information sent to the database driver.
- A prompt flag (*fDriverCompletion*) that governs the policies used when prompting for data source information.

©Copyright 1995-2003 CASEMaker Inc.

NOTE *The connection flow when using* SQLDriverConnect *is the same as the connection flow described in* Table5. *First, you must allocate environment and connection handles. The* SQLDriverConnect *function can then be called to connect to a data source.*

THE INPUT CONNECTION STRING

The input connection string specifies the information needed to connect to a data source.

➲ **Prototype**

Keyword value pairs:

```
KEYWORD=VALUE;
```

The most commonly used keywords are:

- *DSN* — The name of data source name
- *UID* — The username
- *PWD* — The password

➲ **Example**

```
DSN=TEST_DB; UID=myname; PWD=abc;

DSN=TEST DB; UID=myname;

UID=myname;
```

NOTE *If an input connection string has more than one* DSN, UID, *or* PWD, DBMaker *will use the first one.*

USING SQLDRIVERCONNECT WITHOUT A DMCONFIG.INI

The SQLDriverConnect can also be used without a dmconfig.ini file being present for the client site. DBMaker will use default values for each keyword if no keyword(s) are specified in the connection SQLDriverConnect() connection string and no dmconfig.ini is available for the client site.

In the prototype for SQLDriverConnect mentioned above, the argument 'szConnStrIn' is used as the connection string; including new keywords.

©Copyright 1995-2003 CASEMaker Inc.

➲ **Example**

```
SQLDriverConnect (hdbc, hwin, "DSN=TEST DB; UID=SYSADM;PWD=x123
                        ;SVADR=172.0.0.1;
                    PTNUM=12345;ATCMT=0;CTIMO=2;", SQL_NTS, .);
```

THE PROMPT FLAG

The prompt flag indicates whether the Driver Manager or the DBMaker driver needs to use a dialog box to get connection information from the user.

➲ **Example**

Possible values for the prompt flag:

```
SQL_DRIVER_PROMPT
SQL_DRIVER_COMPLETE
SQL_DRIVER_COMPLETE_REQUIRED
SQL_DRIVER_NOPROMPT
```

When the prompt flag value is set to **SQL_DRIVER_COMPLETE** or **SQL_DRIVER_COMPLETE_REQUIRED**, *Driver Manager* performs these actions:

- If the *DSN* is specified in the input connection string, it copies the input connection string and passes the string to the driver.
- If the *DSN* is not specified in the input connection string, Driver Manager displays the *Data Source* dialog box to let the user choose a data source.
- Driver Manager constructs the data source name returned from the dialog box and any other *UID* or *PWD* values found in the input connection string, and passes the string to the driver.

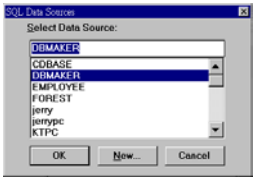

Table6: Data source dialog box

©Copyright 1995-2003 CASEMaker Inc.

If the data source name returned from the dialog box is empty, the Driver Manager specifies the keyword-value pair **DSN=Default** (if a Default data source section exists in ***ODBC.INI***)

➲ **The DBMaker database driver performs the actions listed below based on the following conditions:**

1. If the input connection string contains enough information (user ID and password), the driver connects to the data source.
2. If successful, it copies the input connection string to the output connection string.
3. If the user ID, password, or both are missing, the DBMaker driver displays a dialog box to allow users to fill in the values missing from the input connection string.
4. Connects to the data source after the user leaves the dialog box
5. Constructs a connection string from the value of the DSN in the input connection string and the information returned from the dialog box.Then places the connection string in the output connection string.

Table7: UID and PWD dialog box

NOTE *When prompt flag is set to* SQL_DRIVER_PROMPT*, the* DBMaker *driver behaves the same way as when set to* ***SQL_DRIVER_COMPLETE*** *or* ***SQL_DRIVER_COMPLETE_REQUIRED****. The ODBC Driver Manager will always use a dialog box to prompt for the data source whether you provide a* DSN *in the input string or not.*

Multiple Connections

You can easily connect to more than one data source simultaneously in an application. However, some applications want to connect to the same database multiple times. For example, a task (process) may have two windows, and each window has a connection to the same database. One window is used to scan one table while the other is used to

©Copyright 1995-2003 CASEMaker Inc.

update another table. The SQLConnect command in DBMaker allows a program to connect to the same data source multiple times, but all connections must use the same username and all database changes associated with the connections must belong to the same active transaction.

Example

To connect to data source *DB1* twice by using two valid handles with one user (*user1, pass1*):

```
retcode = SQLAllocConnect(henv, &hdbc1);
retcode = SQLAllocConnect(henv, &hdbc2);

retcode = SQLConnect(hdbc1, "DB1", SQL_NTS, "user1", SQL_NTS, "pass1",
                     SQL_NTS);
retcode = SQLConnect(hdbc2, "DB1", SQL_NTS, "user1", SQL_NTS, "pass1",
                     SQL_NTS);
```

However, if you try to use two users (***user1***, ***pass1***) and (***user2***, ***pass2***) to connect to the same data source in one process, an error will be returned.

Multiple connections to the same data source in a process are not necessary. The more appropriate way to handle this case is to have multiple handle statements under one connection handle.

NOTE *If you want to ensure that each new connection is treated as a separate connection (without merging), set* ***DB_DIFCO=0*** *in the* ***dmconfig.ini****.* For more information about *DB_DIFCO, please refer to the* "Database Administrator's Reference".

©Copyright 1995-2003 CASEMaker Inc.

3.4 Connect Options

A connection to a data source has many attributes that govern its behavior. For example: the **SQL_AUTOCOMMIT** option determines whether all database operations are automatically committed.

SQLSetConnectOption

Each option has a default value defined by the system, but you can use SQLSetConnectOption to specify different values for a connection.

➲ **Prototype**

SQLSetConnectOption:

```
RETCODE SQLSetConnectOption (
                              HDBC    hdbc,
                              UWORD   fOption,
                              UDWORD  vParam);
```

Where hdbc is a valid connection handle, fOption is the option to be set for the connection and vParam is the value specified for fOption.

In DBMaker, the default value for auto-commit mode is on.

➲ **Example**

To turn off the auto-commit mode of a connection:

```
retcode = SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT,
                                    SQL_AUTOCOMMIT_OFF);
```

The option value **SQL_AUTOCOMMIT_OFF** is the new value for the option **SQL_AUTOCOMMIT**. With **SQL_AUTOCOMMIT** set to off, an explicit SQLTransact (**hdbc, COMMIT**) call is required to commit all the changes of a transaction.

©Copyright 1995-2003 CASEMaker Inc.

NOTE *You can use SQLSetConnectOption to set connection options before or after making a connection. These options will remain in effect while the connection handle exists. Once these options are set, they apply to all statements that are associated with the connection. The only exception in* DBMaker *is that the value of option* SQL_CONNECT_MODE *must be set before you make a connection.*

SQLGetConnectOption

When the current value of a connection option is needed, you can use SQLGetConnectOption to get the value.

Prototype

SQLGetConnectOption:

```
RETCODE SQLGetConnectOption (
                              HDBC     hdbc,
                              UWORD    fOption,
                              PTR      vParam)
```

Where hdbc is a valid connection handle, fOption is the option whose value you want to retrieve and vParam points to a location to receive the option value.

Example

To place the value associated with SQL_AUTOCOMMIT for connection hdbc in the variable commitval:

```
retcode = SQLGetConnectOption(hdbc, SQL_AUTOCOMMIT, &commitval);
```


©Copyright 1995-2003 CASEMaker Inc.

3.5 Freeing Handles

Before terminating your application, you should free all resources allocated for connections and the environment. For each of the SQLConnect, SQLAllocConnect, and SQLAllocEnv functions, the corresponding SQLDisconnect, SQLFreeConnect, and SQLFreeEnv functions allow the user to free the corresponding allocated resource.

SQLDisconnect

SQLDisconnect closes the connection associated with a specific connection handle and frees all statement handles under the connection. (Statement handles will be discussed in *Chapter 4*.)

➲ **Prototype**

SQLDisconnect:

```
RETCODE SQLDisconnect(HDBC hdbc)
```

SQLFreeConnect

After closing the connection, a program should call SQLFreeConnect to release the connection handle and free all memory associated with the handle. If you try to use SQLFreeConnect to free an open connection handle, the driver will return an error. You need to close a connection (by calling SQLDisconnect) before calling SQLFreeConnect.

➲ **Prototype**

SQLFreeConnect:

```
RETCODE SQLFreeConnect(HDBC hdbc)
```

©Copyright 1995-2003 CASEMaker Inc.

SQLFreeEnv

SQLFreeEnv frees the environment handle and releases all associated memory. Before calling SQLFreeEnv, an application must call SQLFreeConnect to free any hdbc allocated under the henv.

Prototype

SQLFreeEnv:

```
RETCODE SQLFreeEnv(HENV henv)
```


©Copyright 1995-2003 CASEMaker Inc.

4 SQL Statements

This chapter describes in detail how to use ODBC functions to execute the SQL statements supported by DBMaker. It introduces the SQL query language, and then illustrates how to do the following.

- Allocate and free statement handles by using the functions *SQLAllocStmt* and *SQLFreeStmt*.
- Execute an SQL statement directly by using the function *SQLExecDirect*, and prepare an SQL statement for execution and execute the prepared statement by using the functions *SQLPrepare* and *SQLExecute*.
- Return the number of rows affected by UPDATE, INSERT, or DELETE statements using the function *SQLRowCount*.
- Use parameters to pass a data value to an SQL command at execution time, instead of at preparation time.
- Return the number of parameters in an SQL statement by using the function *SQLNumParams*, and return the description of a parameter marker associated with a prepared SQL statement by using the function *SQLDescribeParam*.
- Bind a buffer to a parameter marker in an SQL statement by using the function *SQLBindParameter*, and input large data items in smaller pieces by using the functions *SQLPutData* and *SQLParamData*.
- Return the current setting of a statement option by using the function *SQLGetStmtOption*, and cancel statement processing by using the function *SQLCancel*.

©Copyright 1995-2003 CASEMaker Inc.

The following diagram shows the topics of Sections 4-3 ~ 4-6 and their relation to the state transitions that occur when writing an application that uses ODBC to access a database.

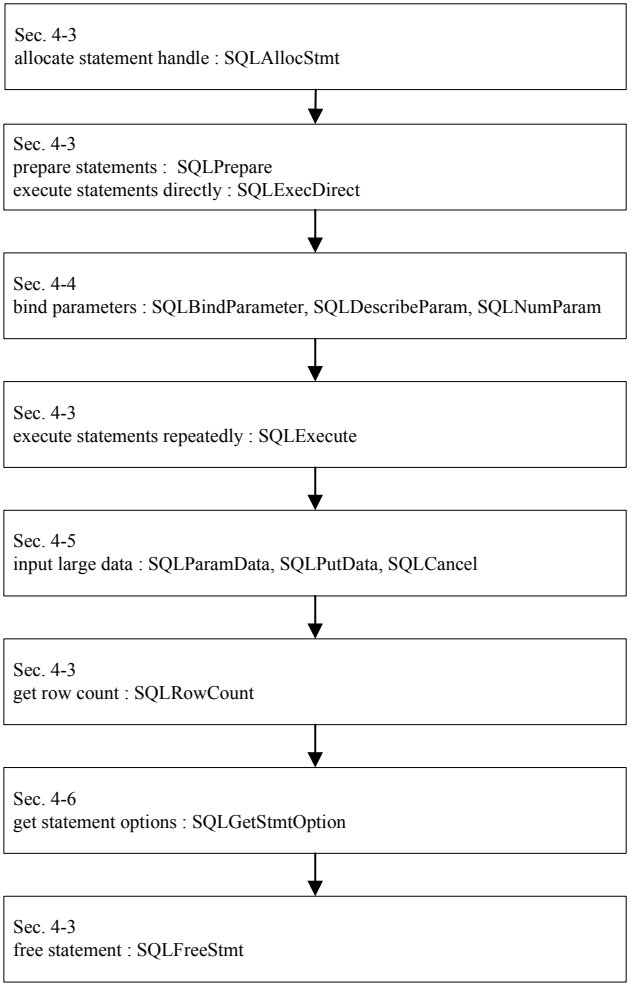

Table8: Topics in this chapter and their relation to state transitions

©Copyright 1995-2003 CASEMaker Inc.

4.1 The SQL Language

Structured Query Language (SQL) is the industry standard query language used for defining, organizing, managing, and retrieving data stored in relational databases. Unlike traditional procedural programming languages such as C and Pascal, you do not need to explicitly define how to perform a database operation. You can simply enter a request to the database using the English-like SQL syntax, and the database will determine the best method to process the request and return the results to you when it is finished.

The functions provided by SQL go beyond simple data retrieval, although that is still one of its most important functions. SQL is actually divided into three parts, known as *Data Definition Language (*DDL*)*, *Data Manipulation Language (*DML*)*, and *Data Control Language (*DCL*)*. Each of these performs a specific role, and together you can use them to perform all functions a DBMS provides, including:

- ***Data definition*** — lets you define the structure and organization of data and the relationships between data.
- ***Data manipulation*** —allows you to retrieve existing data from the database and update the database by adding new data, deleting old data, and modifying previously stored data.
- ***Data control*** —allows you to protect data against unauthorized access, and define integrity constraints to protect data from corruption.

This section provides a brief overview of SQL. For more information, see the "*SQL Command and Function Reference*".

The Role of SQL with ODBC

When the ODBC driver gets an SQL statement, it passes the SQL request to the database engine. The database engine structures, stores and retrieves the data on disk according to the SQL statement.

©Copyright 1995-2003 CASEMaker Inc.

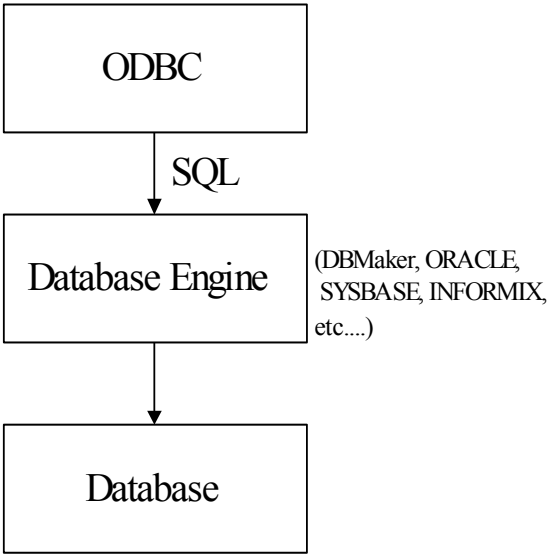

Table9: The role of SQL when using ODBC

Basic SQL Statements

SQL statements can be divided into DDL, DML, and DCL statements. For a more detailed discussion of the entire SQL language and SQL syntax supported by DBMaker, see the "SQL Command and Function Reference".

Data Definition Language (DDL)

The schema of a database is handled by a set of SQL statements called the SQL *Data Definition Language (*DDL*)*. DDL makes use of the CREATE, DROP or ALTER commands to define, remove or modify the definition of a database object. We will briefly explain the CREATE TABLE statement.

©Copyright 1995-2003 CASEMaker Inc.

THE CREATE TABLE COMMAND

A database contains many tables, and each table in a database stores information. Tables are composed of *rows* (records) and *columns* (fields). You can use the CREATE TABLE statement to create a new table in a database.

➲ **Example 1**

Basic syntax for the CREATE TABLE statement:

```
CREATE TABLE table-name (column-name data-type, .... )
```

The ANSI/ISO SQL standard specifies a minimal set of data types that a DBMS should support. Almost all commercial SQL products support these data types or provide similar data types that have equivalent functionality.

➲ **Example 2**

The CREATE TABLE command:

```
CREATE TABLE account (
                      no        serial,   /* account number         */
                      lname     name,     /* account last name      */
                      fname     name,     /* account first name     */
                      branch    integer,  /* belong to branch       */
                      balance   money,    /* account balance        */
                      altno     char(12),
                      stamp     image,    /* account's stamp image  */
                      photo     image,    /* account's photo image  */
                      memo      text      /* account's memo          /
                      );
```

Data Manipulation Language (DML)

Retrieving or manipulating the data in a database is handled by a set of SQL statements called the SQL Data Manipulation Language, or DML. The basic DML statements are SELECT, INSERT, DELETE and UPDATE.

©Copyright 1995-2003 CASEMaker Inc.

RETRIEVING DATA FROM THE DATABASE (SELECT)

You can use the SELECT statement to retrieve data from a database and return a result set to the user.

➲ **Example 1**

Basic syntax of the SELECT statement:

```
SELECT item_list FROM table_list WHERE search_condition;
```

Data Type	Description
CHAR(len)	Fixed-length character string
VARCHAR(len)	Variable-length character string
BINARY(len)	Binary data
OID	Object ID
FILE	BLOB object (file)
LONG VARCHAR	BLOB object (text)
LONG VARBINARY	BLOB object (binary)
SERIAL [(integer)]	Auto-increment integer
SMALLINT	Small integer number
INTEGER [INT]	Integer number
FLOAT	Low-precision floating point number
DOUBLE	High-precision floating point number
DECIMAL [DEC]	Decimal numbers (use default precision and scale)
DECIMAL(precision,scale)	Default precision is 17 and default scale is 6
DATE	Date
TIME	Time
TIMESTAMP	Timestamp
Other	Domain

Table 10: Data types in DBMaker

©Copyright 1995-2003 CASEMaker Inc.

The basic SELECT statement has three components: SELECT, FROM and WHERE. The functions of each of these components are listed below:

- ***SELECT*** — specifies the columns or calculated columns to be retrieved by the query.
- ***FROM*** — specifies the tables that contain the items in the *SELECT* list.
- ***WHERE*** — specifies the search condition that must be met to select a row.

The WHERE clause may contain multiple search conditions which can include:

- Comparison operators (=, >, <, >=, <=, <>, !=)
- Ranges (BETWEEN and NOT BETWEEN)
- Lists (IN and NOT IN)
- String matches (LIKE and NOT LIKE)
- BLOB matches (MATCH and NOT MATCH)
- Unknown values (IS NULL and IS NOT NULL)
- Logical combinations *(AND, OR)*
- Negations *(NOT)*

➲ **Example 2**

Perform a query to find all customers whose account balance is greater than $10,000:

```
select lname, fname, balance from account where balance > 10000
```

MODIFYING DATA IN THE DATABASE

Adding, deleting or updating rows can modify the data contained in a database. For each of these operations, the DBMS will return the number of affected rows.

ADDING DATA TO THE DATABASE

The INSERT statement is used to add a new row to a table.

©Copyright 1995-2003 CASEMaker Inc.

➲ **Example 1**

Basic syntax of the INSERT statement:

```
INSERT INTO table_name(column_names) VALUES value_list
```

The INSERT statement is made up of two components, INSERT INTO and VALUES.

The functions of these components are:

- *INSERT INTO* — specifies the table you want to insert a row into. It can optionally contain a column list to specify that data should only be inserted into those columns. Columns not in this list will be inserted with **NULL** values.
- *VALUES* — specifies the data value you want to insert. You can insert values by using constants or parameters.

As stated above, the value list may contain *constants* or *parameters*. A constant is any numeric, text or date value that can be expressed in text form, such as 'John', 'Monday', 123, 54.823, etc. An example of the INSERT command using constants is shown below.

➲ **Example 2**

Add a new account for John Smith to the database:

```
INSERT INTO account (lname,fname,branch,balance)
       VALUES ('john','smith',101,10000)
```

Parameter data is represented by a question mark (?) in the value list, and values can be inserted later. Parameters can be used when the data values are unknown at preparation time, or when you want to save preparation time. An example of the INSERT command using parameters is shown below. This example is used to insert rows into the database, but the values are not currently known.

➲ **Example 3**

Actual values to be inserted can be bound before execution:

```
INSERT INTO account VALUES (lname,fname,branch) VALUES (?,?,?)
```


©Copyright 1995-2003 CASEMaker Inc.

NOTE *To learn how to prepare statements and bind parameters to ODBC functions, please refer to Section 4 of this chapter.*

DELETING DATA FROM THE DATABASE

The DELETE statement deletes one or more rows from a table.

➲ **Example 1**

Basic syntax of the DELETE statement:

```
DELETE FROM table_name WHERE search_condition
```

The DELETE statement is made up of two components, DELETE FROM and WHERE.

The functions of these components are:

- *DELETE FROM* — specifies the table you want to delete rows from.
- *WHERE* — specifies the search conditions that must be met to delete a row.

The WHERE clause used in a DELETE statement may contain any of the search conditions that are allowed for the WHERE clause used in a SELECT statement.

➲ **Example 2**

Delete the account for John Smith from the database:

```
DELETE FROM account where fname = 'john' and lname = 'smith'
```

UPDATING THE DATA IN THE DATABASE

The UPDATE statement changes data in existing rows in a table.

➲ **Example 1**

Basic syntax of the UPDATE statement is:

```
UPDATE table_name SET column_names expression WHERE search_condition
```

The UPDATE statement has three components, UPDATE, SET, and WHERE.

The functions of these components are:

- *UPDATE* — specifies the table you want to update rows in.

©Copyright 1995-2003 CASEMaker Inc.

- *SET* — specifies the columns you want to change and an expression that defines the changes to be made for each column.
- *WHERE* — specifies the search conditions that must be met to update a row.

The WHERE clause under the UPDATE statement may contain any of the search conditions that are allowed for the WHERE clause under the SELECT statement

➲ **Example 2**

Add 6% interest to all accounts with a balance greater than $1000:

```
UPDATE account SET balance = balance * 1.06 where balance > 1000
```

NOTE *To find out how many rows have been inserted, deleted, or updated, refer to the SQLRowCount function.*

©Copyright 1995-2003 CASEMaker Inc.

4.2 Executing SQL Statements

This section will serve as a guide for writing a simple ODBC program. As the previous section mentioned, every SQL statement can be executed via ODBC in a program. For example, suppose we have connected to a database successfully.

➲ **Example 1**

Use the following SQL statements to construct a simple table:

```
CREATE TABLE account (lname name, fname name, branch integer)

INSERT INTO account VALUES('Mulder', 'Fox', 11240)
```

➲ **Example 2**

The corresponding ODBC statements would be:

```
retcode = SQLAllocStmt(hdbc, &htmt);

retcode = SQLExecDirect(htmt,"CREATE TABLE account (lname name, fname
                              name, branch integer)", SQL_NTS);

retcode = SQLExecDirect(htmt,"INSERT INTO account VALUES('Mulder',
                             'Fox', 11240)", SQL_NTS);
```

SQLAllocStmt

All ODBC functions that use SQL statements need a statement handle. A statement handle is a pointer to a location in the system control area (part of the DCCA) where all information about an SQL statement resides. Therefore, before executing an SQL statement via SQLExecDirect, we need to use SQLAllocStmt to allocate a statement handle.

➲ **Prototype**

SQLAllocStmt:

```
RETCODE SQLAllocStmt(HDBC hdbc, HSTMT FAR *phstmt)
```

©Copyright 1995-2003 CASEMaker Inc.

If the return code is **SQL_SUCCESS**, you have successfully allocated a valid statement handle from the driver. You can then proceed to the next ODBC function, SQLExecDirect.

SQLExecDirect

SQLExecDirect is used to execute an SQL statement directly. Many ODBC books call this *direct execution*, as opposed to another method of execution known as *prepared execution*. Prepared execution will be explained later.

Prototype

SQLExecDirect:

```
RETCODE SQLExecDirect(
                      HSTMT     hstmt,
                      UCHAR FAR *szSqlStr,
                      SWORD     cbSqlStr)
```

The first argument, hstmt, is a valid statement handle, the second argument is the SQL statement string to be executed, and the last argument is the string length of the SQL statement or **SQL_NTS** if szSqlStr points to a null terminated string.

The execution of SQLExecDirect is performed in two phases. First, it compiles (prepares) the SQL statement by checking referenced object names and grammar, chooses an access plan, and converts the statement into an internal executable form. Then in the second phase, it executes the executable form to actually access the database.

If the SQL statement is a query like SELECT * FROM account, then a result set of selected rows is produced and you need to use SQLFetch to get the returned data row by row from the result set (See *Chapter 5*). If the SQL statement is an INSERT, DELETE, or UPDATE statement, SQLRowCount can be used to see how many rows were affected.

©Copyright 1995-2003 CASEMaker Inc.

SQLRowCount

This function returns the number of rows affected by INSERT, DELETE, or UPDATE statements executed in the statement handle.

➲ **Prototype**

SQLRowCount:

```
RETCODE SQLRowCount(
                    HSTMT      hstmt,
                    SDWORD FAR *pcrow)
```

If hstmt is associated with an UPDATE statement, the **pcrow** will return the number of updated rows after executing the UPDATE statement.

➲ **Example**

Using SQLRowCount:

```
SDWORD count;
SDWORD retcode;
retcode = SQLAllocStmt(hdbc, &hstmt);
retcode = SQLExecDirect(hstmt,"CREATE TABLE account (lname name,
                        fname name, branch integer, balance money)",
                        SQL_NTS);
/* insert three records into account table                          */
retcode = SQLExecDirect(hstmt, "INSERT INTO account VALUES('Mulder',
                        'Fox', 11240, 10000.00)", SQL_NTS);
retcode = SQLExecDirect(hstmt, "INSERT INTO account VALUES('Scully',
                        'Dana', 11330, 20000.00)", SQL NTS);
retcode = SQLExecDirect(hstmt, "INSERT INTO account VALUES('Skinner',
                        'Walter', 11240, 30000.00)", SQL_NTS);
/* if branch is 11240, add 1000 to balance                              */
retcode = SQLExecDirect(hstmt, "UPDATE account SET balance = balance +
1000.00 WHERE branch = 11240", SQL_NTS);
```

©Copyright 1995-2003 CASEMaker Inc.

```
/* get the number of updated rows from count in the example.          */
/* Count will be two.                                                 */
retcode = SQLRowCount(hstmt, &count);
```

If hstmt is not associated with an INSERT, DELETE, or UPDATE statement, the row count will be -1 for DBMaker.

SQLFreeStmt

You can use SQLFreeStmt to close or drop a statement handle.

Prototype

SQLFreeStmt:

```
RETCODE SQLFreeStmt (
                     HSTMT     hstmt,
                     UWORD     fOption)
```

The first argument (hstmt) is a valid statement handle and the second is an option that specifies how the statement handle is freed. Two commonly used options for freeing the statement handle are **SQL_CLOSE** and **SQL_DROP**. If the statement is not a select statement, a statement handle can be reused.

Example 1

A reused statement handle:

```
SQLExecDirect(hstmt1, "INSERT ...");
SQLExecDirect(hstmt1, "CREATE ...");
SQLExecDirect(hstmt1, "INSERT ...");
```

If the statement is a select statement, you need to close the statement handle before using it again.

Example 2

Closing a select statement before reusing it:

```
retcode = SQLExecDirect(hstmt,"SELECT * FROM account", SQL_NTS);
...
```


©Copyright 1995-2003 CASEMaker Inc.

```
retcode = SQLFreeStmt(htmt,SQL_CLOSE);

retcode = SQLExecDirect(hstmt,"INSERT INTO account VALUES('Mulder',
                                 'Fox', 11240)", SQL_NTS);
```

By using **SQL_CLOSE** to close a statement handle for reuse later, you do not have to allocate a new statement handle every time you want to execute a statement after a selection. However, if in doubt, you can use the **SQL_DROP** option to drop the statement handle and allocate a new one. Drop will release all resources associated with the statement handle. After a drop, you cannot reuse the handle.

SQLPrepare and SQLExecute

As described in the section on SQLExecDirect, prepared execution is performed in two parts: preparation and execution. If you want to execute a statement repeatedly, you can use prepared execution to improve performance.

Prepared execution divides the execution life of a statement into two parts using the ODBC function calls for preparation (SQLPrepare) and execution (SQLExecute). The idea is to prepare the statement into executable form only once and then execute it many times.

➲ **Prototype**

SQLPrepare:

```
RETCODE SQLPrepare(
                    HSTMT     hstmt,
                    UCHAR     *szSqlStr,
                    UDWORD     cbSqlStr)
```

➲ **Prototype**

SQLExecute:

```
RETCODE SQLExecute(HSTMT     hstmt)
```

In SQLPrepare, hstmt is a valid statement handle, szSqlStr is the SQL statement string to be executed, and cbSqlStr is the length of the string szSqlStr or **SQL_NTS** if the

©Copyright 1995-2003 CASEMaker Inc.

string is null-terminated. Prepared execution is most useful when combined with parameters, which is the topic of the next section.

©Copyright 1995-2003 CASEMaker Inc.

4.3 Parameters

In this section, we will introduce *parameters*, which are used in SQL statements to pass a data value to an SQL command at execution time in an ODBC application program. The concept is similar to the host variables used in embedded SQL.

A parameter is used in an SQL statement when:

- Values of the parameters are unknown at preparation time.
- Applications need to execute the same SQL statement several times with different parameter values (For example, an application may need to use character strings to get all input values, then insert these values into tables in the database. In this case, it can use parameter markers to accept all values, then convert these values to their corresponding column types so the driver can insert them into the database correctly.)
- Applications need to convert the parameter values between different data types.
- Stored procedures need to be executed with the output arguments.

➲ **Example**

To insert five rows into a table named *account*:

```
INSERT INTO account (lname, fname, branch) VALUES (?,?,?)
```

In this statement, '?' is the parameter marker. By using parameters, the application only needs to prepare this statement once, and then execute the prepared statement five times with different parameter values.

Parameter Functions

There are three ODBC functions for dealing with parameters, SQLBindParameter, SQLDescribeParam, and SQLNumParams.

- ***SQLBindParameter***- is used to bind a storage location to a parameter marker and specify the data type, precision, and scale of the storage location.

©Copyright 1995-2003 CASEMaker Inc.

- ***SQLNumParams*** -is used by an application to get the number of parameters in an SQL statement. It is especially useful for an interactive dynamic SQL application.
- ***SQLDescribeParam***- is used to describe the attributes of a specified parameter, like length or precision. It is also used for dynamic SQL applications.

SQLBINDPARAMETER

➲ **Prototype**

SQLBindParameter:

```
RETCODE SQLBindParameter(
              HSTMT      hstmt,
              UWORD      ipar,
              SWORD      fParamType,
              SWORD      fCType,
              SWORD      fSqlType,
              UDWORD     cbColDef,
              SWORD      ibScale,
              PTR        rgbValue,
              SDWORD     cbValueMax,
              SDWORD FAR *pcbValue)
```

An application needs to pass the following information to SQLBindParameter:

- *hstmt* — statement handle
- *ipar* — i^{th} parameter
- *fParamType* — parameter type, input/output
- *fCType* — parameter host language type
- *fSQLType* — SQL column type
- *cbColDef* — precision of the column
- *ibScale* — scale of the column

©Copyright 1995-2003 CASEMaker Inc.

- *rgbValue* — storage address
- *cbValueMax* —the length of storage buffer. When the parameter type is output, this field is in use. The value will be ignored if the returned data has a fixed length in C, such as an integer value. It is not used when the parameter type is input.
- *pcbValue* — length of the parameter in rgbValue

The following example illustrates how several rows of data with different values can be inserted into a database by using SQLBindParameter to bind the row values to parameters before SQLExecute is called. Note that when SQLExecute is called for the third time, a **NULL** will be inserted into the ***branch*** column by setting pcbValue in SQLBindParameter to **SQL_NULL_DATA**.

➲ **Example**

Using SQLBindParameter:

```
#define LENGTH 18
UCHAR  lname[LENGTH], fname[LENGTH];
UDWORD branch_no;
SDWORD retcode, cblname, cbfname, cbbranch;
retcode = SQLPrepare(hstmt,"INSERT INTO account (lname,fname,branch)
                            VALUES (?,?,?)",SQL_NTS);
err_exit(hstmt, retcode);            /* exit if error                */
cblname = SQL NTS;                   /* null terminated string       */
cbfname = 0;
cgbranch = 0;
retcode = SQLBindParameter(hstmt, 1, SQL PARAM INPUT, SQL C CHAR,
                           SQL CHAR, LENGTH, 0, lname, 0, &cblname);
retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, LENGTH, 0, fname, 0, &cbfname);

retcode = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_LONG,
```

©Copyright 1995-2003 CASEMaker Inc.

```
                                SQL_INTEGER, 0, 0, branch_no, 0, &cbbranch);
strcpy(lname, "Mulder");
cblname = strlen(lname);
strcpy(fname, "Fox");
cbfname = strlen(fname);
branch_no = 11240;
retcode = SQLExecute(hstmt);
strcpy(lname, "Scully");
cblname = strlen(lname);
strcpy(fname, "Dana");
cbfname = strlen(fname);
branch_no = 11251;
retcode = SQLExecute(hstmt);
/* insert a NULL data to branch column for the third customer whose   */
/* branch number is unknown                                            */
strcpy(lname, "Angus");
cblname = strlen(lname);
strcpy(fname, "MacGyver");
cbfname = strlen(fname);
cbbranch = SQL_NULL_DATA;      /* indicate NULL for branch column */
retcode = SQLExecute(hstmt);
```

➲ **To set a parameter value:**

1. Call SQLBindParameter to bind a storage location to a parameter marker.
2. Put the parameters value in the storage location.

NOTE *The first step can be done before or after calling SQLPrepare, but should be done before calling SQLExecute. The second step should be done before SQLExecute since the driver needs the parameter values to execute the SQL statement.*

Three parameter types can be used for fParamType in SQLBindParameter:

- *SQL_PARAM_INPUT* - this parameter is called input parameter.

©Copyright 1995-2003 CASEMaker Inc.

- *SQL_PARAM_INPUT_OUTPUT* - this parameter is used for both input and output.
- *SQL_PARAM_OUTPUT* - this parameter is called output parameter.

HOW TO USE SQLBINDPARAMETER WITH INPUT PARAMETERS

The parameter is stored in the storage location rgbValue. When putting the parameter values into rgbValue, you should use the C data types specified in the fCType argument of SQLBindParameter.

The pcbValue argument in SQLBindParameter is simply a pointer to a buffer that contains the parameter length, but it can be used for other purposes.

Possible values stored in pcbValue before calling SQLExecute or SQLExecDirect are:

- The length of the parameter, only useful for character or binary *C* data.
- *SQL_NTS* - to indicate that the parameter value is a null-terminated string.
- *SQL_NULL_DATA*- to indicate that the parameter value is **NULL**, as shown in the previous code example.
- *SQL_DEFAULT_PARAM*- to indicate the default value of the column will be used.
- *SQL_DATA_AT_EXEC* or *SQL_LEN_DATA_AT_EXEC*- to indicate the data for the parameter will be sent with *SQLPutData*. This will be covered in more detail in section *4.4*.

SQLNUMPARAMETER

➲ **Prototype**

SQLNumParams:

```
RETCODE SQLNumParams(
                     HSTMT       hstmt,
                     SWORD  FAR *pcpar)
```

When this function is called, the driver will put the number of parameters in the SQL statement in the buffer pcpar. This number will be zero if the SQL statement contains

©Copyright 1995-2003 CASEMaker Inc.

no parameters. Note that this function can only be called after the SQL statement is prepared (i.e. SQLPrepare has been called).

SQLDESCRIBEPARAM

➲ **Prototype**

SQLDescribeParam:

```
RETCODE SQLDescribeParam(
                         HSTMT       hstmt,
                         UWORD       ipar,
                         SWORD  FAR *pfSqlType,
                         UDWORD FAR *pcbColDef,
                         SWORD  FAR *pibScale,
                         SWORD  FAR *pfNullable)
```

If an application has all the information necessary for SQLBindParameter, then it can call SQLBindParameter directly. However, applications may lack detailed information about the parameters before calling SQLBindParameter to set them. For example, in dynamic SQL applications such as a graphical query tool, the SQL statements to be used are undetermined until runtime. This kind of application needs to obtain detailed information about parameters at one time so that it can bind the parameters.

SQLDescribeParam returns the description of the specified parameter marker associated with a prepared SQL statement.

©Copyright 1995-2003 CASEMaker Inc.

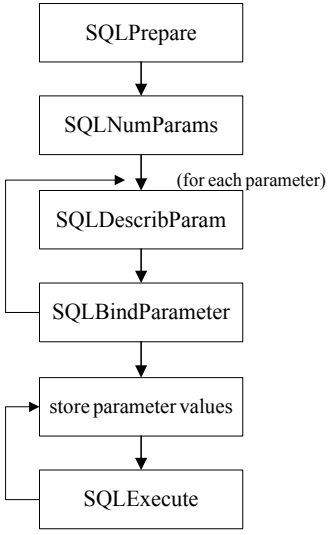

Table12: Program flow when using input parameters.

The following example illustrates the use of SQLDescribeParam, SQLNumParams, and SQLBindParameter in a dynamic SQL application. The parameter marker number is ordered sequentially from left to right, starting at 1.

➲ **Example 1**

Using input parameters in a dynamic SQL application:

```
#define  BUFFER_LEN  256        /* length of the SQL string buffer   */
#define  MAX PARAMS   32        /* allowed max number of parameters  */
UCHAR   str[BUFFER_LEN];
SDWORD  retcode;
SWORD   i, nparam;
SWORD   partype[MAX PARAMS], parscale[MAX PARAMS], parnull[MAX PARAMS];
SWORD   parCtype[MAX_PARAMS];
UDWORD  parlen[MAX_PARAMS], outlen[MAX_PARAMS];
char    *parbuf[MAX PARAMS];
BEGIN:                            /* begin label                     */
```

©Copyright 1995-2003 CASEMaker Inc.

```
getSQLString(&str);                /* get input SQL statement string  */
retcode = SQLPrepare(hstmt,str,SQL NTS);    /* prepare the input SQL string
*/
err_exit(hstmt, retcode);          /* exit if error                   */
retcode = SQLNumParams(cmdp,&nparam); /* get number of parameters     */
err exit(hstmt, retcode);          /* exit if error                   */
if (nparam > 0)                    /* parameters found in input string*/
   {
    printf("There are %d parameters \n",nparam);
    for (i = 0; i < nparam; i++)  /* describe parameters and set them*/
        {
         retcode = SQLDescribeParam(cmdp, i+1, &partype[i], &parlen[i],
                                     &parscale[i], &parnull[i]);
         err exit(hstmt, retcode);          /* check return code, exit if
error*/
         /* allocate storage location for the parameter according to the
*/
         /* parameter type, length and scale, reuse the storage if possible
*/
         allocParamStorage(partype[i], parlen[i], parscale[i], &parbuf[i]);
         /*get C type corresponding to SQL type                       */
         getSQLCtype(partype, &parCtype);
         /* bind the parameter to storage location                    */
         retcode = SQLBindParameter(cmdp, i+1, SQL_PARAM_INPUT,
                                          parCtype[i], partype[i],
                                          parlen[i], parscale[i],
                                          parbuf[i], BUFFER_LEN,
                                          &outlen[i]);
         err exit(hstmt, retcode);/* check return code, exit if error*/
        }
   for (i = 0; i < nparam; i++)
       {
```


©Copyright 1995-2003 CASEMaker Inc.

```
        /* get parameter values and store them in bound storage
*/

        getParamValue(nparam, parCtype[i], partype[i], parlen[i],
parscale[i],

                      &parnull[i], parbuf[i]);

       }

   }                              /* end of if statement            */

retcode = SQLExecute(hstmt);      /* excute prepared SQL statement   */

err_exit(hstmt, retcode);

if (user Quit())                  /* user wants to quit              */

   return;

else

   goto BEGIN;                    /* go to BEGIN for next SQL string */
```

Only when a stored procedure with output arguments is executed do we need to use output parameters. Later we will demonstrate how to use the output parameters to execute a stored procedure with output arguments. A stored procedure is a user-defined function. Once the stored procedure is created, it is stored in an executable format in the database as an object of the database. You can execute a stored procedure as a command in the interactive SQL window, or you can invoke it in an application program, a trigger action, or another stored procedure. Here we only describe how to execute a stored procedure in ODBC application programs. For more information about stored procedures, please refer to the “Database Administrator's Reference”.

The following example illustrates how to call the SQLBindParameter function with the SQL_PARAM_OUTPUT parameter type. A buffer is prepared by the SQLBindParameter function, which stores the value returned when a stored procedure is executed with a specific input city value like: Taipei.

©Copyright 1995-2003 CASEMaker Inc.

➲ **Example 2**

Using the output parameter:

```
SDWORD personNumber;

SDWORD retcode, avlen;

retcode = SQLPrepare(hstmt,"CALL getNumber("Taipei",?)",SQL NTS);

err_exit(hstmt, retcode);         /* exit if error                  */

retcode = SQLBindParameter(hstmt, 1, SQL_PARAM_OUTPUT, SQL_C_LONG,

                           SQL INTEGER, 0, 0, &personNumber, 0,

                           &avlen);
err_exit(hstmt, retcode);         /* exit if error                  */

retcode = SQLExecute(hstmt);      /* excute prepared SQL statement   */

printf("total %ld employees live on Taipei \n", personNumber);
```

If you want to write a dynamic SQL program with output parameters, you may use SQLNumParams, SQLDescribeParam, SQLProcedures, and SQLProcedureColumns. You can use SQLProcedures to get the list of procedure names that are stored in the data source. Then use SQLProcedureColumns to retrieve information about procedure parameters.

Using Parameters in SQLExecDirect

As stated before, when an application wants to execute an SQL statement more than once, it may call SQLPrepare first, then call SQLExecute several times instead of preparing the same SQL statement for each execution.

Parameters can also be used in an SQL statement that is executed only once by using SQLExecDirect. However, you must bind the parameters and set the parameter values before calling SQLExecDirect anyway, so you lose all of the advantages you get when using parameters with SQLPrepare and SQLExecute.

There is no need to use parameters with SQLExecDirect unless the data input is of a special type, such as a BLOB. Use the **SQL_DATA_AT_EXEC** or the **SQL_LEN_DATA_AT_EXEC** options to bind the parameter at execution time for a BLOB; the data values are not set until after the statement is executed.

©Copyright 1995-2003 CASEMaker Inc.

➲ **Example**

Using parameters with SQLExecDirect:

```
#define  LENGTH   18
UCHAR   lname[LENGTH], fname[LENGTH];
UDWORD  branch no;
SDWORD  retcode, cblname, cbfname, cbbranch;
retcode = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, LENGTH, 0, lname, 0, &cblname);
retcode = SQLBindParameter(hstmt, 2, SQL PARAM INPUT, SQL C CHAR,
                           SQL CHAR, LENGTH, 0, fname, 0, &cbfname);
retcode = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_LONG,
                           SQL_INTEGER, 0, 0, branch_no, 0, &cbbranch);
strcpy(lname, "Bill");
cblname = strlen(lname);
strcpy(fname, "Skinner");
cblname = strlen(fname);
branch no = 11243;
retcode = SQLExecDirect(hstmt, "INSERT INTO account (lname, fname,
```

branch) VALUES (?,?,?)", SQL_NTS);

```
err_exit(hstmt, retcode);
```

Clearing Bound Parameters

After a storage location is bound by calling SQLBindParameter, it can be reused repeatedly. In the example, three storage locations are bound to the three-parameter markers in the INSERT statement and remain bound until they are explicitly unbound.

The storage locations are unbound when the application calls SQLFreeStmt with the **SQL_RESET_PARAMS** option or the **SQL_DROP** option. Notice that the three storage locations belong to the same statement handle, and when SQLFreeStmt is called, all the storage locations bound in the statement handle will be unbound. If the

©Copyright 1995-2003 CASEMaker Inc.

application uses the **SQL_RESET_PARAMS** option, it can reset the statement handle and bind a different storage location.

➲ **Example**

Clearing bound parameters:

```
#define  LENGTH  18
UCHAR  lname [LENGTH], fname[LENGTH];
UDWORD branch_no;
SDWORD retcode, cblname, cbfname, cbbranch;
retcode = SQLBindParameter(hstmt, 1, SQL PARAM INPUT, SQL C CHAR,
                           SQL_CHAR, LENGTH, 0, lname, 0, &cblname);
retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL_CHAR, LENGTH, 0, fname, 0, &cbfname);
retcode = SQLBindParameter(hstmt, 3, SQL PARAM INPUT, SQL C LONG,
                           SQL_INTEGER, 0, 0, branch_no, 0, &cbbranch);
... (use the three parameters to execute some SQL commands)
/* reset the parameters */
retcode = SQLFreeStmt(hstmt, SQL_RESET_PARAMS);
```

The handle will be released, and become invalid, when an application uses the **SQL_DROP** option in the **SQLFreeStmt** statement.

©Copyright 1995-2003 CASEMaker Inc.

4.4 Entering Large Data

DBMaker provides two methods to input BLOB data into a database. One method uses a small, fixed size buffer to read a portion of the BLOB data and enter it into the database. By repeating this process several times, the entire BLOB can be input without having to use a single large buffer. DBMaker also provides a File Object data type to allow users to store BLOB data in an external file.

How to Enter Large Data

When you need to enter large amounts of data into a long varchar or long varbinary column, you can use the SQLPutData and SQLParamData functions to enter the data in smaller segments. Thus, a large buffer is not needed to store all of the data, as it would be if the data were entered all at once.

➲ **Prototype**

SQLParamData:

```
RETCODE SQLParamData(
                      HSTMT      hstmt,
                      PTR   FAR  *prgbValue)
```

➲ **Prototype**

SQLPutData:

```
RETCODE SQLPutData(
                    HSTMT   hstmt,
                    PTR     rgbValue,
                    SDWORD  cbValue)
```

SQLParamData is used to check if parameters require data. Data is then entered using SQLPutData. This process continues until data has been entered for all of the parameters in the SQL statement.

©Copyright 1995-2003 CASEMaker Inc.

➲ **To enter large data objects into a database**

1. Bind the parameters — Set the pcbValue argument in the SQLBindParameter function to either SQL_DATA_AT_EXEC or SQL_LEN_DATA_AT_EXEC. This lets the ODBC driver know that you will provide values for this parameter at execution time using SQLPutData.

2. Execute the SQL command — Execute the SQL statement with SQLExecDirect or SQLExecute. When there are parameters that are expecting data at execution time, the call will return SQL_NEED_DATA.

3. Find the first parameter expecting data at execution time — Call SQLParamData to indicate that the first parameter expecting data at execution time should start to receive data.

4. **Call SQLPutData** — Prepare the next segment of data in the buffer and call SQLPutData to send it to the database for the parameter currently waiting for data. Repeat this step until all of the data for this parameter is sent.

5. **Call SQLParamData** — If the return code is SQL_NEED_DATA, the next parameter that is expecting data at execution time is ready to receive data, and you should go back to Step 4. If the return code is SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, then all data for all parameters expecting data at execution time has been sent and the SQL statement has completed its execution.

©Copyright 1995-2003 CASEMaker Inc.

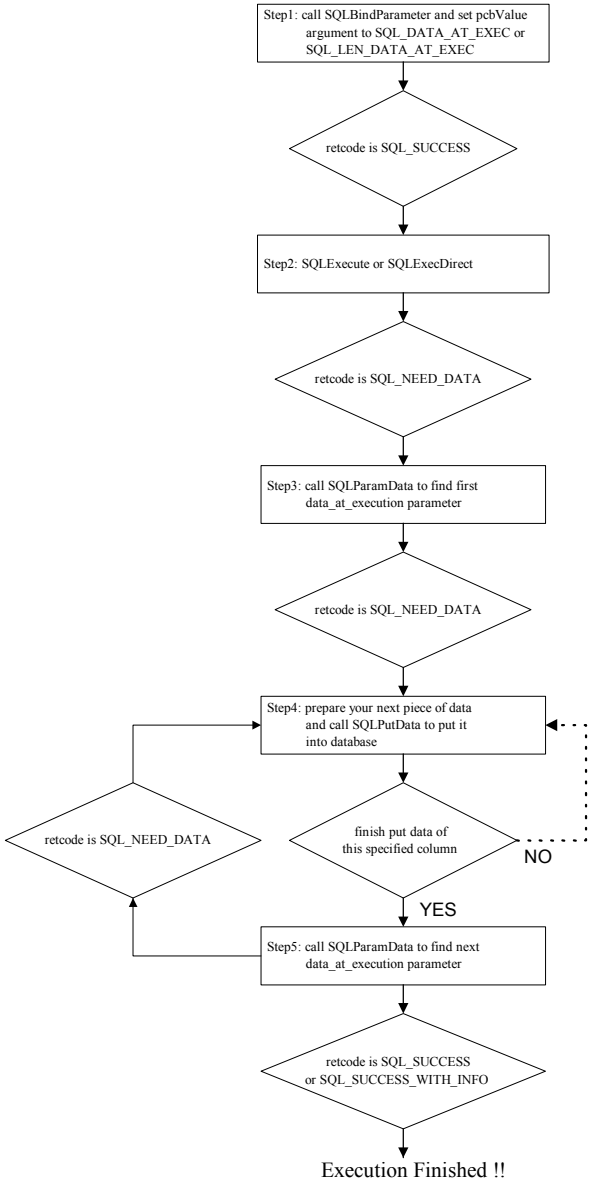

Table13: Process flow for entering large data

©Copyright 1995-2003 CASEMaker Inc.

In the following example, we insert a record into the ***account*** table. **InitUserData**() opens a customer's data file that includes the customer's ***photograph***, ***signature*** and a ***memo*** field to be entered into the ***account*** table. **GetUserData**() retrieves the next **MAX_DATA_SIZE** bytes of data from the users data file into the data buffer InputData until all data has been read. **SQLPutData** sends data from the buffer to the database.

➲ **Example**

Inserting data with SQLParamData and SQLPutData:

```
#define  MAX_DATA_SIZE  2048


SDWORD  cbPhoto, cbStamp, cbMemo;
SDWORD  DataLen;
PTR     pParm, DataFile;
UCHAR   InputData[MAX DATA SIZE];
SDWORD  retcode;
retcode = SQLPrepare(hstmt, (UCHAR *)"INSERT INTO account
                            VALUES('Mark','Greene',2, 40000.00,
                             'xxxx', ?, ?, ?)",SQL NTS);
if (retcode == SQL_SUCCESS)
   {
    /* Bind the parameters. Set cbPhoto, cbStamp and cbMemo with     */
    /* SQL DATA AT EXEC to let ODBC know that values of these parameters */
    /*  will be provided at statement execution time      */
    cbPhoto = cbStamp = cbMemo = SQL_DATA_AT_EXEC;
    retcode = SQLBindParameter(hstmt, 1, SQL PARAM INPUT, SQL C BINARY,
                               SQL LONGVARBIANRY, 0, 0, NULL, 0,
                               &cbPhoto);
    retcode = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_BINARY,
                               SQL LONGVARBIANRY, 0, 0, NULL, 0,
                               &cbStamp);
```


©Copyright 1995-2003 CASEMaker Inc.

```
retcode = SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_CHAR,
                           SQL LONGVARCHAR, 0, 0, NULL, 0,
                           &cbMemo);
retcode = SQLExecute(hstmt);
if (retcode == SQL_NEED_DATA)  /* any large data ? */
    {
     Parm = 0;
     while ((retcode = SQLParamData(hstmt, &pAddr)) == SQL_NEED_DATA)
            {          /* need to put large data for this column */
             Parm++;   /* in a loop to get data and put data in blob */
             InitUserData(Parm, DataFile);
             while (GetUserData(Parm, DataFile, InputData, &DataLen))
                    retcode = SQLPutData(hstmt, InputData, DataLen);
            }
     if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
         printf("insert a record of account table success !! \n");
    }
}
```

©Copyright 1995-2003 CASEMaker Inc.

Canceling the Execution of SQLPutData

If an error occurs while entering data into the database or we decide not to continue entering data, we can cancel the process by using the SQLCancel function.

➲ **Prototype**

SQLCancel:

```
RETCODE SQLCancel(HSTMT hstmt)
```

You can call SQLCancel and the whole statement will be aborted. After canceling the current execution of a statement, you can call SQLExecute or SQLExecDirect again.

Placing Large Data in a File Object

A *File Object (FO)* is a powerful large object data type supported by DBMaker. A FO is similar to **LONG VARCHAR** or **LONG VARBINARY** data, but it is stored as an external file on your host file system. Defining a column with the **FILE** data type creates an FO column.

➲ **Example 1**

Defining a *photograph* column as a FO column:

```
create table student (name char(20), photograph file)
```

Since an FO is treated as BLOB data, you can use BLOB insertion methods to insert data into FO columns. By setting fSQLType to **SQL_LONGVARCHAR** or **SQL_LONGVARBINARY**, DBMaker will create a new file for each FO column and copy data from the input buffer (when fCType is **SQL_C_CHAR**) or from a file on the client site (when fCType is **SQL_C_FILE**).

Instead of creating another file, you may want to link an FO column to an existing file on the server side, such as a file on a CD-ROM. If you want to link a server side file, set fCType to **SQL_C_CHAR** and fSqlType to **SQL_FILE**. When using this method, copy the file name into the buffer before calling SQLExecute.

©Copyright 1995-2003 CASEMaker Inc.

FSQL Type	FO Column	FC Type	Data Source
SQL_LONGVARCHAR or SQL_LONGVARBINARY (Same as BLOB)	Creates a new file on the server side.	SQL_C_FILE	Copies data from a file on the client site.
		SQL_C_CHAR SQL_C_BINARY	Copies data from the input buffer.
SQL_FILE	Links to an existing file on the server side.	SQL_CHAR	Gets the file name from the input buffer.

In the following example, a new record is inserted for a customer named *Mary* along with her photo, which is stored as a File Object. The file already exists on the server side, so we link this file into the database.

➲ **Example 2**

Placing large data in a File Object:

```
UCHAR  pPhotoFlName[80];

SDWORD retcode;
SDWORD cbPhoto;

retcode = SQLPrepare(hstmt, "INSERT INTO student VALUES('Mary', ?)",

                     SQL_NTS);

CbPhoto = SQL NTS;

retcode = SQLBindParameter(hstmt, 1, SQL PARAM INPUT, SQL C CHAR,

                           SQL_FILE, 80, 0, pPhotoFlName, 0, &cbPhoto);

strcpy(pPhotoFlName, "/disk1/sys/fo/photo");       /* pass the file name */


retcode = SQLExecute(hstmt);
```

Since an FO is stored as an external file, applications used to edit the file can still work on the file directly.

©Copyright 1995-2003 CASEMaker Inc.

4.5 Get and Set Options

The current setting of a statement option in a statement handle can be obtained by using the SQLGetStmtOption function.

➲ **Prototype**

SQLGetStmtOption:

```
RETCODE SQLGetStmtOption(
                        HSTMT  hstmt,
                        UWORD  fOption,
                        PTR    pvParam);
```

➲ **Prototype**

SQLSetStmtOption:

```
RETCODE SQLSetStmtOption{
                          HSTMT   hstmt,
                          WORD    fOption,
                          UDWORD  vParm);
```

Where hstmt is a valid statement handle, fOption is the option to be retrieved and pvParam is the value associated with fOption. Depending on the value of fOption, a 32-bit integer value or a pointer to a null-terminated character string will be returned in pvParam.

DBMaker provides some options that can be used with these two functions.

Option	Description
SQL_GET_INCREMENT_VALUE	Gets the value of a SERIAL column.
SQL_GET_CURRENT_OID	Gets the OID of the most recently inserted/fetched tuple.
SQL_GET_BACKUP_ID	Gets the backup ID.
SQL_DUMP_PLAN	Gets dump plan options.
SQL_PLAN	Gets the plan for the query execution.
SQL_PLAN_LEN	Gets the plan length.

©Copyright 1995-2003 CASEMaker Inc.

Table 14 Extended statement options used with SQLGetStmtOption

Option	Description	Permitted Values
SQL_DUMP_PLAN	Sets dump plan options .	SQL_DUMP_PLAN_ON SQL_DUMP_PLAN_OFF

Table 16 Extended statement options used with the SQLSetStmtOption

In the following example, the table ***account*** contains a **SERIAL** data type column. The value of a **SERIAL** column is incremented automatically, and users do not need to explicitly specify it. However, after inserting a record, users may want to know the value of the **SERIAL** column just inserted. This value can be obtained by calling SQLGetStmtOption with fOption **SQL_GET_INCREMENT_VALUE**.

➲ **Example 1**

Getting the value of a SERIAL column using SQLGetStmtOption:

```
/* insert a record into table account where the value of each field     */
/* is its default value                                                 */
SDWORD val;
SDWORD retcode;
retcode = SQLExecDirect(hstmt, "INSERT INTO ACCOUNT VALUES ()",
                        SQL_NTS);
/* get the serial number that was just inserted                          */
retcode = SQLGetStmtOption(hstmt, SQL_GET_INCREMENT_VALUE, &val);
```

In this example, the variable **val** will be the value of the **SERIAL** column that was just inserted into the table account in the previous INSERT statement. For the definition and the use of the **SERIAL** data type, please refer to the “SQL Command and Function Reference”.

Another special use of SQLGetStmtOption is to get the OID of the most recently inserted or fetched tuple. Continuing from the previous example, you can submit SQLGetStmtOption with fOption **SQL_GET_CURRENT_OID** to get the OID of the record just inserted into the database.

©Copyright 1995-2003 CASEMaker Inc.

➲ **Example 2**

Getting the current object's OID using SQLGetStmtOption:

```
UCHAR  oid[8];
SDWORD retcode;
/* insert a record into account table */
retcode = SQLExecDirect(hstmt, "INSERT INTO ACCOUNT VALUES ()",
                        SQL_NTS);
/* get the OID of the record just inserted */
retcode = SQLGetStmtOption(hstmt, SQL_GET_CURRENT_OID, &oid);
```

An OID is a unique ID for an object in DBMaker. You can use the OID to uniquely specify an object in a database.

➲ **Example 3**

Using OID in the WHERE clause of a query:

```
SELECT * FROM account WHERE OID = ?
```

NOTE *Refer to the* "SQL Command and Function Reference" *or the* "Database Administrator's Reference", *for more detailed information on the OID data type.*

The extended statement options **SQL_DUMP_PLAN**, **SQL_PLAN_LEN**, and **SQL_PLAN** are used to get the query plan generated by the DBMaker Query Optimizer for a prepared SQL statement.

➲ **To get the plan of a prepared SQL statement**

1. Turn on the SQL_DUMP_PLAN option by calling SQLSetStmtOption.
2. Get the length of the plan string by calling SQLGetStmtOption with fOption SQL_PLAN_LEN
3. Allocate a buffer according to the plan string and then call SQLGetStmtOption with fOption SQL_PLAN to get the plan string.

➲ **Example 4**

Getting an SQL statement's query plan using the SQLSetStmtOption and SQLGetStmtOption:

©Copyright 1995-2003 CASEMaker Inc.

```
SDWORD planlen;
UCHAR  *planstr;
SDWORD retcode;
/* turn on the dump plan option                                          */
retcode = SQLSetStmtOption(hstmt, SQL_DUMP_PLAN, SQL_DUMP_PLAN_ON);
/* prepare an SQL JOIN statement                                         */
retcode = SQLPrepare(hstmt, "SELECT * FROM account, branch WHERE
                     account.branchId = branch.branchId", SQL_NTS);
/* get the plan string length                                            */
retcode = SQLGetStmtOption(hstmt, SQL PLAN LEN, &planlen);
/* allocate a buffer for the plan string according to the plan           */
/* string length                                                         */
planstr = malloc(planlen);
/* get the plan string                                                   */
retcode = SQLGetStmtOption(hstmt, SQL_PLAN, planstr);
```

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.

5 Retrieving Results

Performing queries to retrieve data is one of the most important functions of a database.

In this chapter, we examine how to perform the following:

- Retrieve data on a row-by-row basis by binding columns to a storage location using the functions *SQLBindCol* and *SQLFetch*.
- Obtain information (such as type and length) about the columns in a result set by using the functions *SQLNumResultCols*, *SQLDescribeCol,* and *SQLColAttributes*.
- Use a cursor to execute a positioned *DELETE* or positioned *UPDATE* on a result set obtained from a query. We also examine how to get or set the cursor name by using the functions *SQLGetCursorName* and *SQLSetCursorName*.
- Retrieve large data objects ***(LONG VARCHAR** or **LONG VARBINARY)*** and file objects piece by piece by using the function *SQLGetData*. Retrieving large objects or file objects piece by piece allows you to use much smaller buffers than would be required to retrieve the data all at once.
- Use rowsets to browse forward or backward through a result set obtained by querying with the functions *SQLExtendedFetch* and *SQLSetPos*.

©Copyright 1995-2003 CASEMaker Inc.

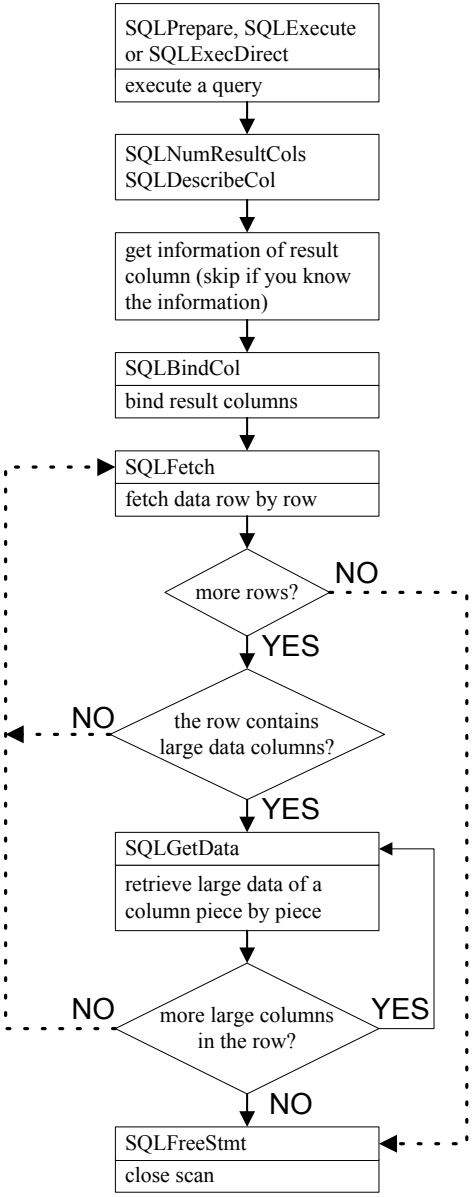

Table18: Program flow for retrieving data from a database.

©Copyright 1995-2003 CASEMaker Inc.

5.1 Queries Using ODBC

When an application needs to retrieve data from a database, the most common method is to perform a query by using the SQL SELECT statement. In this section, how to perform a query and fetch the result data, row by row using ODBC functions is covered.

Binding Storage Locations and Fetching Data

Suppose we want to get last name, first name, and branch information for customers at branch 11240 from the ***account*** table.

➲ **Example**

Query:

```
SELECT lname, fname, branch FROM account WHERE branch = 11240
```

After preparing and executing this query, we are ready to fetch the data row by row. If all the information in the result columns in the projection of this query is known (e.g. column type, precision, scale, etc.), then we can use SQLBindCol and SQLFetch to fetch the results.

- ***SQLBindCol***- is used to associate a storage location with a column of data. The role of *SQLBindCol* in a *SELECT* statement is similar to that of the *SQLBindParameter* in an *INSERT* statement.
- ***SQLFetch***- is used to fetch a row of data from the result set. The driver will return data for all bound columns to the storage locations specified by *SQLBindCol*.

©Copyright 1995-2003 CASEMaker Inc.

➲ **Prototype**

SQLBindCol:

```
RETCODE SQLBindCol(
                    HSTMT       hstmt,
                    UWORD       icol,
                    SWORD       fCType,
                    PTR         rgbValue,
                    SDWORD      cbValueMax,
                    SDWORD FAR *pcbValue)
```

An application needs to pass the following information to SQLBindCol so it can associate the storage location with the result column.

- *fCType* — the data type to which the data is to be converted.
- *rgbValue* — the address of an output buffer for the data. The application must allocate this buffer and make sure the buffer is large enough to accommodate the data retrieved for the specified data type.
- *cbValueMax* — the length of the output buffer. This value is ignored if the returned data has a fixed width in C, such as an integer value.
- *pcbValue* — the address of a storage buffer that is used to return the number of bytes of available data. Note that the driver will store ***SQL_NULL_DATA*** in this argument if the fetched data is ***NULL***.

After each column in the projection is bound, SQLFetch() is called to fetch a row of data.

➲ **Prototype**

SQLFetch:

```
RETCODE SQLFetch(HSTMT hstmt)
```

The following example uses ODBC to perform the query shown earlier with SQLBindCol and SQLFetch.

©Copyright 1995-2003 CASEMaker Inc.

➲ Example

To fetch all customers information at branch 11240 from the ***account*** table:

```
#define  LENGTH  18

UCHAR       lname[LENGTH], fname[LENGTH];
UDWORD      branch no, TRUE = 1;
SDWORD      retcode, cblname, cbfname, cbbranch;
retcode = SQLExecDirect(hstmt, "SELECT lname, fname, branch FROM
                                       account WHERE branch = 11240",
                                       SQL NTS);
if (retcode == SQL_SUCESS)
   {
    retcode = SQLBindCol(hstmt,1, SQL C CHAR, lname, LENGTH, &cblname);
    retcode = SQLBindCol(hstmt,2, SQL C CHAR, fname, LENGTH, &cbfname);
    retcode = SQLBindCol(hstmt,3, SQL_C_LONG, &branch_no, 0, &cbbrach);
   }
/* fetch data one row at a time and print out the result data        */
/* stop when no more data or error returned from SQLFetch            */
while (TRUE)
   {
    retcode = SQLFetch(hstmt);
    if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
        {
         if (cblname == SQL_NULL_DATA)            /* check null data */
            printf("last name: NULL\n");
         else
            printf("last name: %s\n", lname);
         if (cbfname == SQL_NULL_DATA)
            printf("first name: NULL\n");
         else
```

©Copyright 1995-2003 CASEMaker Inc.

```
            printf("first name: %s\n", fname);
         if (cbbranch == SQL NULL DATA)
            printf("branch no: NULL\n");
         else
            printf("branch no: %d\n", branch_no);
        }
    else                        /* if no more data or errors returned */
         break;
   }
```

Result Columns Characteristics

Some applications may not know what data types will be inserted in advance. Likewise, it is possible that in dynamic SQL an application does not know the result data that it will fetch ahead of time. If this is the case, the SQLNumResultCols and SQLDescribeCol functions can help provide more information. SQLNumResultCols is used to get the number of result columns in the result set. SQLDescribeCol is used to describe characteristics of a result column, including the name, SQL type, precision, scale, and whether or not the column allows **NULL** values.

➲ **Prototype**

SQLNumResultCols:

```
RETCODE SQLNumResultCols(
                          HSTMT       hstmt,
                          SWORD  FAR *pccol)
```

➲ **Prototype**

SQLDescribeCol:

```
RETCODE SQLDescribeCol(
                        HSTMT       hstmt,
                        UWORD       icol,
                        UCHAR  FAR *szColName,
```


©Copyright 1995-2003 CASEMaker Inc.

```
                    SWORD       cbColNameMax,
                    SWORD  FAR *pcbColName,
                    SWORD  FAR *pfSqlType,
                    UDWORD FAR *pcbColDef,
                    SWORD  FAR *pibScale,
                    SWORD  FAR *pfNullable)
```

©Copyright 1995-2003 CASEMaker Inc.

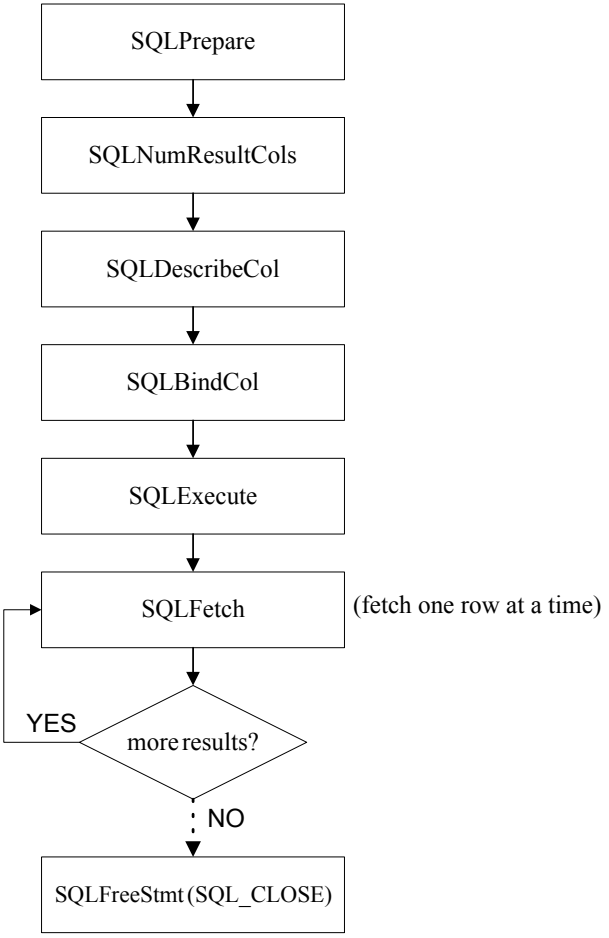

Table19: Program flow for obtaining a result set

After preparing a query, you can call SQLNumResultCols to find out how many result columns are in the query. Next call SQLDescribeCol to get information about how much memory each column will need, which is used in turn used to call SQLBindCol. Finally, the result data can be fetched one row at a time by calling SQLFetch.

©Copyright 1995-2003 CASEMaker Inc.

The example shown below uses SQLNumResultCols, SQLDescribeCol, SQLBindCol, and SQLFetch to fetch a result set after performing a query.

➲ **Example**

Fetching a result set after performing a query:

```
#define  BUFFER LEN  256       /* length of the query string buffer*/
#define  MAX_COLS    128       /* allowed max number of columns */

UCHAR       str[BUFFER_LEN];
SDWORD      retcode, TRUE = 1;
SWORD       i, ncol;
SWORD       coltype[MAX_COLS], colscale[MAX_COLS], colnull[MAX_COLS];
SWORD       colCtype[MAX_COLS];
UDWORD      collen[MAX COLS], outlen[MAX COLS];
char       *colbuf[MAX_COLS];

BEGIN:                          /* begin label                  */

getQueryString(&str);           /* get user input query string   */

/* prepare the input query                                          */
retcode = SQLPrepare(hstmt,queryStr, SQL NTS);

err_exit(hstmt, retcode);       /* exit if error                */

retcode = SQLNumResultCols(hstmt,&ncol); /* get number of result columns */
err_exit(hstmt, retcode);       /* exit if error                */

if (ncol > 0)                   /* still columns in input query  */
   {
    printf("There are %d result columns \n",ncol);

    for (i = 0; i < ncol; i++)    /* describe columns and bind them */
```

©Copyright 1995-2003 CASEMaker Inc.

```
        {
        retcode = SQLDescribeCol(hstmt, i+1, &coltype[i], &collen[i],
                                 &colscale[i], &colnull[i]);
        err_exit(hstmt, retcode); /* exit if error                  */

        /* allocate storage location for column according to its,   */
        /* type, length and scale, reuse the storage if possible    */
        allocColumnStorage(coltype[i], collen[i], colscale[i], &colbuf[i]);

        /* get corresponding SQL C type                             */
        getSQLCtype(coltype, &colCtype);

        /* bind the column storage                                  */
        retcode = SQLBindCol(hstmt, i+1, colCtype[i], coltype[i],
                             collen[i], colscale[i], colbuf[i],
                             BUFFER_LEN, &outlen[i]);
        err_exit(hstmt, retcode);/* exit if error                  */
       }                         /* end of for                     */
  }                              /* end of if                      */

retcode = SQLExecute(hstmt);     /* execute the prepared query     */

err_exit(hstmt, retcode);        /* exit if error                  */

/* fetch one row at a time until no more data is in the result set,*/
/* if the column data is null, add a mark in the column buffer, then output
*/
/* all the column data (print to file or standard output)           */

while(TRUE)
     {
      retcode = SQLFetch(hstmt);
```


©Copyright 1995-2003 CASEMaker Inc.

```
      if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
         {
          for (i = 0; i < ncol; i++)
              {/* if data is NULL, specify NULL in column buffer    */
               if (outlen[i] == SQL_NULL_DATA)
                   MarkNullColumn(colbuf[i]);
               /* output column data                                 */
               outputColumnData(ncol, colCtype[i], coltype[i],
                                      collen[i],   colscale[i],
                                      colnull[i],  colbuf[i]);
              }
         }
      else
         break;
      }

retcode = SQLFreeStmt(hstmt, SQL_CLOSE); /* close the cursor associated  */
                                         /* in the statement hstmt  */

err_exit(hstmt, retcode);                /* exit if error           */

if (user Quit())                         /* user wants to quit      */
   return;
else
   goto BEGIN;                           /* go to BEGIN for next query   */
```

More about Result Columns

Although we can get some of the characteristics of columns by calling SQLDescribeCol, there may still be additional column information that applications need to know. ODBC provides the function SQLColAttributes for this purpose.

©Copyright 1995-2003 CASEMaker Inc.

SQLColAttributes

SQLColAttributes is used to return descriptor information for a column. This information is for the specified descriptor type.

Prototype

SQLColAttributes:

```
RETCODE SQLColAttributes(
                        HSTMT       hstmt,
                        UWORD       icol,
                        UWORD       fDescType,
                        PTR         rgbDesc,
                        SWORD       cbDescMax,
                        SWORD  FAR *pcbDesc,
                        SDWORD FAR *pfDesc)
```

The descriptor types defined in ODBC include:

- SQL_COLUMN_COUNT
- SQL_COLUMN_NAME
- SQL_COLUMN_TYPE
- SQL_COLUMN_LENGTH
- SQL_COLUMN_PRECISION
- SQL_COLUMN_SCALE
- SQL_COLUMN_DISPLAY_SIZE
- SQL_COLUMN_NULLABLE
- SQL_COLUMN_UNSIGNED
- SQL_COLUMN_MONEY
- SQL_COLUMN_UPDATABLE
- SQL_COLUMN_AUTO_INCREMENT

©Copyright 1995-2003 CASEMaker Inc.

- **SQL_COLUMN_CASE_SENSITIVE**
- **SQL_COLUMN_SEARCHABLE**
- **SQL_COLUMN_TYPE_NAME**
- **SQL_COLUMN_TABLE_NAME**
- **SQL_COLUMN_OWNER_NAME**
- **SQL_COLUMN_QUALIFIER_NAME**
- **SQL_COLUMN_LABEL**

NOTE *For detailed information about the meaning of each option, see the "Microsoft ODBC Programmer's Reference".*

For example, if the value of fDescType is **SQL_COLUMN_TYPE**, SQLColAttributes will return the SQL type of the specified column. Although this information can also be obtained by using SQLDescribeCol, SQLColAttributes can also provide other information that cannot be obtained by using SQLDescribeCol.

The major differences between SQLColAttributes and SQLDescribeCol are:

- *SQLDescribeCol* provides some specific values for one column at one time, while *SQLColAttributes* gets only the value of one descriptor.
- *SQLColAttributes* provides more specific and detailed column information. It can also be extended if a driver adds more driver-specific descriptors or if *ODBC* defines new descriptors in future versions.

For example, an application that needs to know if a column is case-sensitive can use the SQLColAttributes function with the descriptor option **SQL_COLUMN_CASE_SENSITIVE** to find out.

➲ **Example**

SQLColAttributes to get detailed column information:

```
#define  TRUE   1
#define  FALSE  0
SDWORD   CSflag;              /* case-sensitive flag                */
```

©Copyright 1995-2003 CASEMaker Inc.

```
SDWORD   retcode;

retcode = SQLPrepare(hstmt, "SELECT lname, fname, branch FROM account
                             WHERE branch = 11240", SQL_NTS);

retcode = SQLColAttributes(hstmt, 1, SQL COLUMN CASE SENSITIVE,
                           NULL, 0, NULL, &CSflag);
if (CSflag == TRUE)
    printf("Column 1 is case-sensitive\n");
```

Clear Bound Columns

After a storage location is bound to a column by calling SQLBindCol, it can be reused repeatedly. In the example, three storage locations are bound to the three columns in the SELECT statement.

➲ **Example**

Call SQLFreeStmt with the **SQL_UNBIND** option to unbind all bound columns associated with a statement handle:

```
Retcode = SQLFreeStmt(hstmt, SQL_UNBIND);
```

Now all of the bound storage locations in hstmt are cleared, and an application can reuse the statement handle and bind a different storage area. If the application wants to unbind a single bound column, an alternative is to call SQLBindCol and pass a **NULL** pointer in the rgbValue argument.

If an application does not need to reuse the statement handle, then SQLFreeStmt can be called with the **SQL_DROP** option. In this case, all the storage locations are cleared and any existing cursors, pending results, and resources used by the statement handle will be freed as well.

©Copyright 1995-2003 CASEMaker Inc.

5.2 Cursors

A cursor is a tool that allows you to step through a result set row-by-row for row-conditional processing. Applications can perform multiple operations on each individual row in a given result set. A cursor is opened on the result set by execution of a query.

When to Use Cursors

A cursor is used when the program needs to perform update or delete operations on specific rows in a result set. For example, the program might retrieve some rows from the query results, display them on the screen for the user, and then respond to a user's request to update or delete data.

If you wish to update data using a cursor, the SELECT statements that are used to generate the result set must explicitly specify FOR UPDATE or FOR UPDATE OF column_list in the SELECT statement. (E.g. SELECT * FROM account FOR UPDATE) If the statement has been not declared with FOR UPDATE, it will default to a read only cursor and you will not be allowed to do cursor updates or deletes.

➲ **Example 1**

UPDATE statements using cursors:

```
UPDATE tablename SET column = value [, column = value...]
       WHERE CURRENT OF cursorname
```

➲ **Example 2**

DELETE statements using cursors:

```
DELETE FROM tablename WHERE CURRENT OF cursorname
```

Getting the Cursor Name

The ODBC driver will automatically generate a name that begins with the letters **SQL_CUR** when you call SQLAllocStmt to allocate a statement handle. You can use

©Copyright 1995-2003 CASEMaker Inc.

SQLGetCursorName to get the full name of the cursor associated with a specific statement handle.

Prototype

SQLGetCursorName:

```
RETCODE SQLGetCursorName(
                        HSTMT      hstmt,
                        UCHAR FAR *szCursor,
                        SWORD      cbCursorMax,
                        SWORD FAR *pcbCursor)
```

Using Cursors

The following shows how to use SQLGetCursorName in positioned updates, which involves two different hstmts for the SELECT and UPDATE statements.

Example

Using cursors in positioned updates to update John Smith's branch number:

```
#define  NAME_LEN    21
#define  CURSOR LEN  20
HSTMT   hstmtSelect;
HSTMT   hstmtUpdate;
UCHAR   szLname[NAME_LEN],szFname[NAME_LEN],cursorName[CURSOR_LEN];
SWORD   cursorLen;
SDWORD  sBranch, cbName, cbBranch;
/* Allocate the statement handles                                    */
retcode = SQLAllocStmt(hdbc, &hstmtSelect);
retcode = SQLAllocStmt(hdbc, &hstmtUpdate);
/* SELECT the result set and bind its columns to local storage       */
/* NOTE:   This is a select FOR UPDATE                               */
retcode = SQLExecDirect(hstmtSelect, "SELECT lname,fname, branch
```


©Copyright 1995-2003 CASEMaker Inc.

```
                                FROM account FOR UPDATE", SQL_NTS);
retcode = SQLBindCol(hstmtSelect, 1, SQL_C_CHAR, szLname, NAME_LEN,
                     &cbLname);
retcode = SQLBindCol(hstmtSelect, 2, SQL_C_CHAR, szFname, NAME_LEN,
                     &cbFname);
retcode = SQLBindCol(hstmtSelect, 3, SQL_C_INTEGER, &sBranch, 0,
                     &cbBranch);
/* get the cursor name of the select for use in the update statement     */
retcode = SQLGetCursorName(hstmtSelect, cursorName, CURSOR_LEN,
                           &cursorLen);
/* Read through the result set until the cursor is positioned on the row */
/* for John Smith                                                        */
do
   retcode = SQLFetch(hstmtSelect);
while ((retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) &&
       (strcmp(szFname, "John") != 0 && strcmp(szLname, "Smith") != 0
       && sbranch == 2100));
/* Perform a positioned update of John Smith's branch number             */
/* NOTE: the cursorName in the CURRENT OF clause                         */
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO
   {
     sprintf(updsql, "UPDATE account SET branch = 2101 WHERE
                      CURRENT OF %s", cursorName);
     retcode = SQLExecDirect(hstmtUpdate, updsql, SQL_NTS);
    }
```

Setting the Cursor Name

You can use SQLSetCursorName to set the cursor name of an active statement handle. You have to use SQLSetCursorName to change the cursor name before executing the SELECT statement.

©Copyright 1995-2003 CASEMaker Inc.

Prototype

SQLSetCursorName:

```
RETCODE SQLSetCursorName(
                         HSTMT      hstmt,
                         UCHAR FAR *szCursor,
                         SWORD      cbCursor)
```


©Copyright 1995-2003 CASEMaker Inc.

5.3 Fetching Large Data

As we described before, the normal way to get column data is to use SQLBindCol to bind a local buffer for the column. During SQLFetch, the column data is automatically stored in the bound buffer. You can use the bind method to retrieve large data if you are sure that your buffer is big enough.

The alternative is to use SQLGetData to get one buffer full of data each time. If using SQLGetData, do not bind the column, otherwise SQLFetch will automatically send the column data to the bound buffer.

An application needs to pass the following information to SQLGetData so it can associate a result column with a storage location:

- *fcType* — the data type to which the data is to be converted.
- *rgbValue* — the address of an output buffer for the data. The application must allocate this buffer and make sure the buffer is large enough to accommodate the data retrieved for the specified data type.
- *cbValueMax* — the length of the output buffer. This value is ignored if the returned data has a fixed width in *C*, such as integer data.
- *pcbValue* — the address of a storage buffer which is used to return the number of bytes of available (remaining) data before the current call to *SQLGetData*.

©Copyright 1995-2003 CASEMaker Inc.

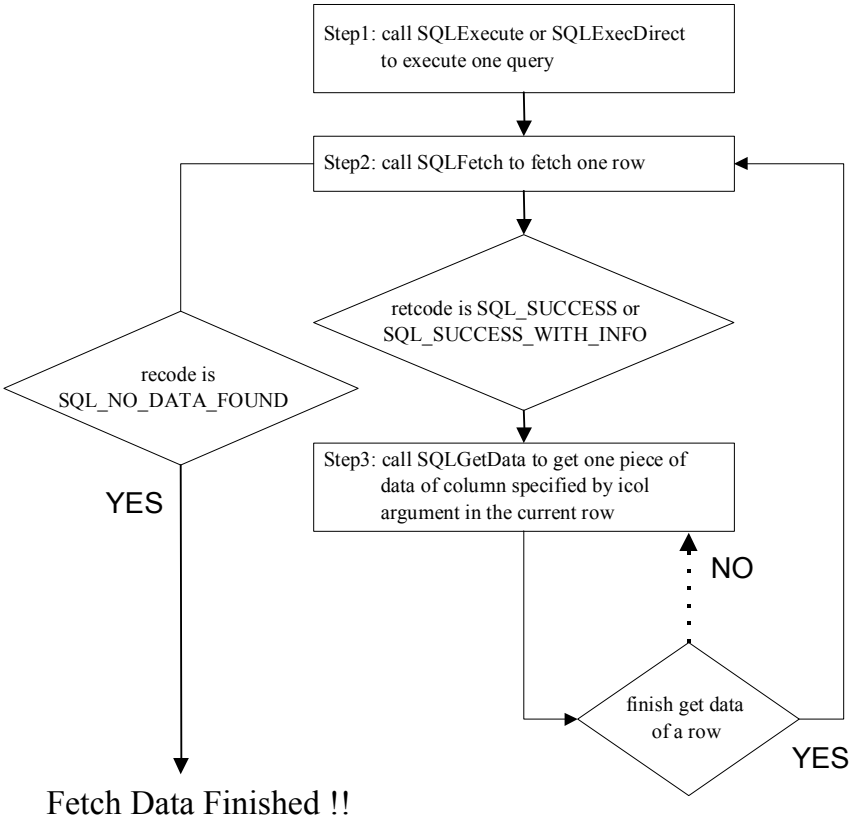

Table20: Program flow for retrieving large data

©Copyright 1995-2003 CASEMaker Inc.

SQLGetData

When you need to retrieve large data from a **LONG VARCHAR** or **LONG VARBINARY** column, you can use the SQLGetData function to retrieve the data segment by segment. This way you do not have to prepare a large buffer to retrieve the whole column.

➲ **Prototype**

SQLGetData:

```
RETCODE SQLGetData(
                    HSTMT hstmt,
                    UWORD icol,
                    SWORD fCType,
                    PTR rgbValue,
                    SDWORD cbValueMax,
                    SDWORD FAR * pcbValue)
```

➲ **To retrieve large data objects from a database**

1. Execute the SQL command — Execute the SQL query with SQLExecDirect or SQLExecute.
2. Fetch the data — call SQLFetch to get the next row of data. If SQLFetch returns SQL_NO_DATA_FOUND, then all rows in the result set of the query have been returned. If the return code is SQL_SUCCESS or SQL_SUCCESS_WITH_INFO and there is large data you want to fetch, go to the next step.
3. Get the large data objects — Call SQLGetData to get one piece of the data in the unbound column specified by the icol argument in the current row. Repeat this step until SQLGetData returns SQL_NO_DATA_FOUND. If you want to fetch data from the next row, go back to the previous step.

In the following example, we will fetch the columns; ***fname***, ***photo,*** and ***memo*** from all rows in the ***account*** table and display them. The ***photo*** and ***memo*** columns contain large objects. We use the bind method to get the values in the ***fname*** column and SQLGetData to get the values for the ***photo*** and ***memo*** columns.

©Copyright 1995-2003 CASEMaker Inc.

As we explained before, you can either bind a column to a storage location or use SQLGetData when retrieving data. This is true for all data, and if you wish, you can use SQLGetData for regular data types such as integers. However, this is not practical because it involves extra programming effort and is not necessary.

Example

```
#define  MAX_BINARY_SIZE  1024
#define  MAX CHAR SIZE     256
#define  MAX_NAME_SIZE      21

SDWORD cbFname, cbPhoto, cbMemo, DataLen;
UCHAR  FnameBuf[MAX NAME SIZE], PhotoBuf[MAX BINARY SIZE],
       MemoBuf[MAX_CHAR_SIZE];
PTR    PhotoDataFile, MemoDataFile;
SDWORD retcode, TRUE=1;
retcode = SQLExecDirect(hstmt, (UCHAR *)"SELECT fname, photo, memo
                                          FROM account", SQL_NTS);
retcode = SQLBindCol(hstmt, 1, SQL_C_CHAR, FnameBuf, MAX_NAME_SIZE,
                     &cbFname);
while (TRUE)
      {
        retcode = SQLFetch(hstmt);

        /* After calling SQLFetch, the value of bound column fname is    */
        /* automatically stored in user buffer FnameBuf             */
if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WTIH_INFO)
           {
            /* InitDataFile() opens user's data files for storing        */
            /* photo and memo column data                                */
            InitDataFile(PhotoDataFile);
            InitDataFile(MemoDataFile);
```


©Copyright 1995-2003 CASEMaker Inc.

```
/* The size of remaining data before this SQLGetData is      */
/* returned in cbPhoto. This SQLGetData will retrieve        */
/* MAX BINARY SIZE data of column photo from the database,   */
/* and put it into binary buffer PhotoBuf.                   */
while(TRUE)
     {
      retcode = SQLGetData(hstmt, 2, SQL_C_BINARY,
                           PhotoBuf,
      MAX_BINARY_SIZE, &cbPhoto);
      if (retcode == SQL SUCCESS
          || retcode == SQL_SUCCESS_WTIH_INFO)
          {
           /* GetToFile() moves data from buffer PhotoBuf    */
           /* to user file PhotoDataFile                     */
           GetToFile(PhotoDataFile, PhotoBuf);
           printf("%ld more bytes remains in Photo column \n",
                  cbPhoto - MAX_BINARY_SIZE);
          }
      else
           break;
      }
/* Use SQLGetData to get memo data and put in MemoDataFile   */
while (TRUE)
      {
       retcode = SQLGetData(hstmt, 3, SQL C CAHR, MemoBuf,
       MAX CHAR SIZE, &cbMemo);
       if (retcode == SQL_SUCCESS
           || retcode == SQL_SUCCESS_WTIH_INFO)
          {
           GetToFile(MemoDataFile, MemoBuf);
```

©Copyright 1995-2003 CASEMaker Inc.

```
                    }
                else
                     break;
               }

          /* Display data on screen                                    */
          Display(FnameBuf);
          DisplayLargeData(PhotoDataFile);
          DisplayLargeData(MemoDataFile);
         }
     else if (retcode == SQL_NO_DATA_FOUND)
         {
          printf("no data found \n");
          break;
         }
     else
        {
         printf("error \n");
         break;
        }
     }
```

For most applications, all of the data in a large column is usually retrieved and stored in a temporary file before it is displayed. While this is still the case for static data that must be displayed all at once, streaming (audio, video) or page (large text) data can be displayed without the need for a temporary file. Using a double buffer scheme to retrieve and display data simultaneously, as shown in the diagram.

©Copyright 1995-2003 CASEMaker Inc.

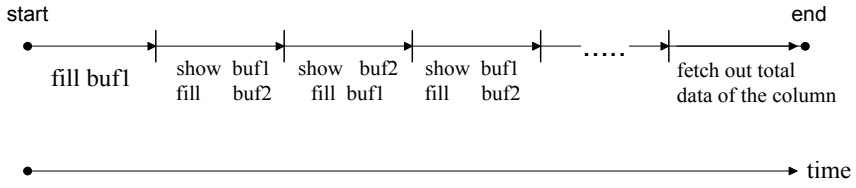

Table21: Using double buffers to get large data

Stopping SQLGetData Operations

If an error occurs when retrieving data from a database, or if you do not want to continue retrieving data, you can stop the retrieval process by using the SQLFreeStmt function with the **SQL_CLOSE** option. You call the SQLFreeStmt function with the **SQL_CLOSE** option to close the cursor and discard all pending results. You can reopen this cursor to retrieve data by calling SQLExecute or SQLExecDirect with the same query again.

Binding Columns to Retrieve File Objects

If you want to retrieve a large data object and have it placed in a client file, you can use the *bind file* method to do it. In this method, you set the fCType argument of SQLBindCol to **SQL_C_FILE** and place the file name in the buffer. This instructs DBMaker to create a file and copy the large object data into it.

The following example uses this method to retrieve a photograph and place it in a client file by binding **SQL_C_FILE** in the fCType argument of SQLBindCol and preparing the filename in the buffer pPhotoFlName. After calling SQLFetch, the photograph is copied into the file.

➲ **Example**

```
UCHAR  pPhotoFlName[80];

SDWORD retcode;

retcode = SQLBindCol(hstmt, 1, SQL C FILE, pPhotoFlName, 80, &cbPhoto);

strcpy(pPhotoFlName, "/disk1/usr/fo/photo");

retcode = SQLExecDirect(hstmt, "SELECT photograph FROM student
```

©Copyright 1995-2003 CASEMaker Inc.

```
                                WHERE name = 'mary'", SQL_NTS);

retcode = SQLFetch(hstmt);  /* a new file is created and data copied */
```

Fetching the Filename of File Objects

As described in chapter four, File Objects (FO) are stored as external files on the server. You can use any of the three methods listed above to retrieve FO data, but the bind client file (**SQL_C_FILE**) method always creates a new file on the client site. If you are only interested in the file name of the FO, you can use the built-in function **FILENAME()** to get the file name.

➲ **Example**

Use **SQL_C_CHAR** to bind the column:

```
UCHAR  pPhotoFlName[80];
SDWORD retcode;

retcode = SQLBindCol(hstmt, 1, SQL_C_CHAR, pPhotoFlName, 80, &cbPhoto);

retcode = SQLExecDirect(hstmt, "SELECT FILENAME(photograph) FROM

                                student WHERE name = 'mary'", SQL_NTS);

retcode = SQLFetch(hstmt);   /* file name of FO goes to pPhotoFlName */
```

Currently, most multimedia tools process multimedia data stored as an operating system file. If the multimedia data is stored in a **LONG VARCHAR** or **LONG VARBINARY** column, you need to fetch the data from DBMaker and redirect it to a file that is accepted by the multimedia tool. If you stored it as a file object, you only need to get the file name from DBMaker and pass the name to the tool.

©Copyright 1995-2003 CASEMaker Inc.

5.4 Manipulating Result Sets

Applications can use SELECT statements to query the underlying database. In addition to the SQLFetch function from the previous sections, DBMaker also allows you to use SQLExtendedFetch to easily browse backward or forward through the result set returned by the SELECT command, and SQLSetPos to further modify the result set from SQLExtendedFetch.

Rowsets

A *rowset* behaves like a window on the result set; we can use it to browse the details of the result set. The rowset is always a subset of the result set and has the same tuple order.

You can fetch a rowset using the SQLExtendedFetch function. However, before calling SQLExtendedFetch you must allocate a buffer, bind the result columns, and set the number of tuples (rowset size) that you want to fetch.

You can call SQLExtendedFetch with different options to move the rowset window backward or forward to any position within the result set. For example, if the rowset size is ten, the option SQL_FETCH_FIRST moves the window to the head of the result set, and reads the first ten tuples into the rowset. Applications are responsible for setting the rowset size by calling the SQLSetStmtOption function with the SQL_ROWSET_SIZE option. The default value for the SQL_ROWSET_SIZE option is one. Applications are also responsible for allocating enough buffer space to bind columns using SQLBindCol before calling SQLExtendedFetch.

Program Flow

The program flow is similar to SQLFetch except for buffer allocation for the rowset. You can change rowset size between SQLExtendedFetch calls, but you must ensure the column output buffer and column status array is large enough for each column. SQLBindCol must be called again to rebind any newly allocated column output buffers and column status arrays. The only exception is that the **rgfRowStatus** array

©Copyright 1995-2003 CASEMaker Inc.

argument of SQLExtendedFetch, which is used to record the status of rows in a rowset and whose size is the same as rowset size, can only be rebound by recalling SQLExtendedFetch.

Storage Binding

You use the SQLBindCol function to bind the output buffer (**rgbValue**) and column status (**pcbValue**) for fetched data (rowset) from the result set. Since the number of tuples fetched can be any value up to the rowset size, you must allocate enough space for the output buffer and column status array with respect to the rowset size. Otherwise, the SQLExtendedFetch function may fail and place output tuple data in an illegal address space.

➲ **Prototype**

SQLBindCol:

```
RETCODE SQLBindCol(
                    HSTMT       hstmt,
                    UWORD       icol,
                    SWORD       fCType,
                    PTR         rgbValue,
                    SDWORD      cbValueMax,
                    SDWORD FAR *pcbValue)
```

There are two ways to bind the output buffer and column status array for a rowset which might have more than one tuple: *column-wise binding* and *row-wise binding*.

COLUMN-WISE BINDING

Use SQLSetStmtOption to set SQL_BIND_TYPE to BIND_BY_COLUMN to specify column-wise binding. If you are using column-wise binding, the buffers for the same column from all tuples of the rowset will be sequential. That is, you allocate enough buffer space for one column at a time. For example, the code fragment to bind columns in a table with two columns (int and char(5)) follows.

NOTE *SQLFetch is a special case of SQLExtendedFetch if column-wise binding is used.*

©Copyright 1995-2003 CASEMaker Inc.

➲ Example

```
#define ROWSET SIZE   6                    /* rowset size(6 tuples) */
#define NAME LEN     30                    /* length of NAME column */
#define AGE_LEN       4                    /* length of AGE column  */
SDWORD  retcode;
char    *c1 rgbValue, *c2 rgbValue;
SDWORD  *c1 pcbValue, *c2 pcbValue;
UWORD   *rgfRowStatus;
UDWORD   crow;
int      irow;
/* set rowset size and exit if there is an error                        */
retcode = SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWSET_SIZE);
err_exit(hstmt, retcode);
/* set the binding type and exit if there is an error                   */
retcode = SQLSetStmtOption(hstmt, SQL_BIND_TYPE, SQL_BIND_BY_COLUMN);
err_exit(hstmt, retcode);
/* allocate buffer for the column data (rowset) and column status arrays */
c1 rgbValue = (char *)malloc(ROWSET SIZE*NAME LEN);     /* c1 data   */
c1_pcbValue = (SDWORD *)malloc(ROWSET_SIZE*sizeof(SDWORD)); /* c1 status */
c2_rgbValue = (char *)malloc(ROWSET_SIZE*AGE_LEN);      /* c2 data   */
c2_pcbValue = (SDWORD *)malloc(ROWSET_SIZE*sizeof(SDWORD)); /* c2 status */
/* allocate row status array for the rowset                             */
rgfRowStatus = (UWORD *)malloc(ROWSET_SIZE*sizeof(UWORD));
/* prepare the input query and exit if there is an error                */
retcode = SQLPrepare(hstmt, "select NAME, AGE from employee", SQL_NTS);
err exit(hstmt, retcode);
/* bind the columns and exit if there is an error                       */
retcode = SQLBindCol(hstmt, 1, SQL_C_CHAR, c1_rgbValue, NAME_LEN,
                     c1_pcbValue);
```

©Copyright 1995-2003 CASEMaker Inc.

```
err_exit(hstmt, retcode);
retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, c2_rgbValue, AGE_LEN,
                     c2_pcbValue);
err_exit(hstmt, retcode);
/* execute the prepared query and exit if there is an error              */
retcode = SQLExecute(hstmt);
err_exit(hstmt, retcode);
/* fetch 6 tuples at once until there is no more data in the result set  */
while(TRUE)
    {
     retcode = SQLExtendedFetch(hstmt, SQL_FETCH_NEXT, 0, &crow,
                                rgfRowStatus);
     if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
         {
          printf("**** %d fetched rows in rowset ****\n", crow);
          /* print tuples in rowset                                    */
          for (irow=0; irow<ROWSET_SIZE; irow++)
              {
               printf("row %d of rowset - ",irow+1);
               switch(rgfRowStatus[irow])
                   {
                    case SQL_ROW_SUCCESS:
                    printf("(NAME: %s) , ",c1_rgbValue + irow * NAME_LEN);
                    printf("(AGE : %s)   ",c2_rgbValue + irow * AGE_LEN);
                    printf("[SUCCESS] \n");
                    break;
                    case SQL_ROW_NOROW:
                    printf(" [NO ROW] \n");
                    break;
                    case SQL_ROW_ERROR:
```


©Copyright 1995-2003 CASEMaker Inc.

```
                    printf("[ROW ERROR] \n");
                    break;
                   }
              }
         }
          else
               break;
    }
/* close cursor associated with hstmt and exit if there is an error      */
retcode = SQLFreeStmt(hstmt, SQL CLOSE);
err_exit(hstmt, retcode);
```

ROW-WISE BINDING

If SQLSetStmtOption is used to set SQL_BIND_TYPE to any value other than BIND_BY_COLUMN, then buffers will be bound row by row. In this case, the value will be used as the length of the necessary output buffer for one tuple. The buffer value is the columns' output value and columns' status for all columns. If the columns are already known, applications often define a structure that is composed of all the columns output buffers and status buffers. For example, the code fragment for a table with 2 columns (int and char(5)) to bind columns follows.

➲ **Example**

```
#define ROWSET_SIZE   6                    /* rowset size(6 tuples) */
#define NAME_LEN     30                    /* length of NAME column */
#define AGE LEN       4                    /* length of AGE column  */
SDWORD  retcode;
char    *c1_rgbValue, *c2_rgbValue, *tup_rgbValue, *tup_prn;
SDWORD  *c1_pcbValue, *c2_pcbValue;
UWORD   *rgfRowStatus;
UDWORD   crow;
int      irow, tup_len;
/* set the rowset size and exit if there is an error                    */
```

©Copyright 1995-2003 CASEMaker Inc.

```
retcode = SQLSetStmtOption(hstmt, SQL_ROWSET_SIZE, ROWSET_SIZE);
err_exit(hstmt, retcode);
/*calculate the length of one row                                    */
tup_len = NAME_LEN + sizeof(SDWORD) + AGE_LEN + sizeof(SDWORD);
/* set the binding type to row-wise and exit if there is an error     */
retcode = SQLSetStmtOption(hstmt, SQL_BIND_TYPE, tup_len);
err_exit(hstmt, retcode);
/* allocate buffer for the column data(rowset) and column status arrays  */
tup_rgbValue = (char *)malloc(ROWSET_SIZE*tup_len);
/* allocate row status array for rowset                               */
rgfRowStatus = (UWORD *)malloc(ROWSET_SIZE*sizeof(UWORD));
/* prepare the input query and exit if there is an error              */
retcode = SQLPrepare(hstmt, "select NAME, AGE from employee", SQL_NTS);
err_exit(hstmt, retcode);
/* bind the columns and exit if there is an error                     */
c1_rgbValue = tup_rgbValue;
c1_pcbValue = (SDWORD *)(c1_rgbValue + NAME_LEN);
c2_rgbValue = c1_rgbValue + NAME_LEN + sizeof(SDWORD);
c2_pcbValue = (SDWORD *)(c2_rgbValue + AGE_LEN);
retcode = SQLBindCol(hstmt, 1, SQL_C_CHAR, c1_rgbValue, NAME_LEN,
                     c1_pcbValue);
err_exit(hstmt, retcode);
retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, c2_rgbValue, AGE_LEN,
                     c2_pcbValue);
err_exit(hstmt, retcode);
/* execute the prepared query and exit if there is an error           */
retcode = SQLExecute(hstmt);
err_exit(hstmt, retcode);
/* fetch 6 tuples at once until there is no more data in the result set  */
while(TRUE)
```


©Copyright 1995-2003 CASEMaker Inc.

```
    {
     retcode = SQLExtendedFetch(hstmt, SQL FETCH NEXT, 0, &crow,
                                rgfRowStatus);
     if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
         {
          tup prn = tup rgbValue;
          printf("**** %d fetched rows in rowset ****\n", crow);
          /* print tuples in rowset                                  */
          for (irow=0; irow<ROWSET_SIZE; irow++)
              {
               printf("row %d of rowset - ",irow+1);
               switch(rgfRowStatus[irow])
                   {
                    case SQL ROW SUCCESS:
                    printf("(NAME: %s) , ",tup_prn);
                    printf("(AGE : %s)   ",tup_prn + NAME_LEN +
sizeof(SDWORD);
                    printf("[SUCCESS] \n");
                    break;
                    case SQL_ROW_NOROW:
                    printf(" [NO ROW] \n");
                    break;
                    case SQL_ROW_ERROR:
                    printf("[ROW ERROR] \n");
                    break;
                   }
               /* print next tuple */
               tup_prn = tup_prn + tup_len;
              }
         }
```

©Copyright 1995-2003 CASEMaker Inc.

```
        else
            break;
    }
/* close cursor associated with hstmt and exit if there is an error      */
retcode = SQLFreeStmt(hstmt, SQL_CLOSE);
err_exit(hstmt, retcode);
```

Positioning the Cursor

SQLExtendedFetch positions the cursor on the first row of a rowset if a cursor exists. SQLExtendedFetch might be used by:

- Positioned *UPDATE* and *DELETE* statements from another statement handle. You can call SQL*ExtendedFetch* to position the cursor on a row and use a positioned *DELETE* statement to delete that row from the result set of the target table. For example, *DELETE … WHERE CURRENT OF …*
- *SQLGetData*. You can call SQL*GetData* to get data for those columns that are not bound. The rowset size should be set to one before calling *SQLGetData*.
- *SQLSetPos* with the *SQL_DELETE, SQL_REFRESH, SQL_UPDATE* options.

Prior to the first time you call SQLExtendedFetch, the cursor is positioned before the start of the result set, which should be seen as undefined. Using different options might position the cursor before the start of result set or after the end of result set, instead of on an existing row.

Arguments of SQLExtendedFetch

Prototype

SQLExtendedFetch:

```
RETCODE SQLExtendedFetch(
                        HSTMT         hstmt,
                        UWORD         fFetchType,
                        SDWORD        irow,
```


©Copyright 1995-2003 CASEMaker Inc.

```
                    UDWORD FAR   *pcrow,
                    UWORD  FAR   *rgfRowStatus)
```

The possible return values from **SQLExtendedFetch** are:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_ERROR
- SQL_INVALID_HANDLE

The following table shows the rowset and return code returned when the ODBC application requests different rowsets. ODBC applications should check the return code from SQLExtendedFetch and the row status for each row of the rowset before it uses the contents of the rowset buffers.

Requested Rowset	Return Code	Cursor Position	Returned Rowset
Before start of result set.	sql_no_data_found	Before start of result set	None. The contents of the rowset buffers are undefined.
Overlaps start of result set.	sql_success	First row of rowset.	First rowset in the result set.
Within result set.	sql_success	First row of rowset.	Requested rowset.

©Copyright 1995-2003 CASEMaker Inc.

Requested Rowset	Return Code	Cursor Position	Returned Rowset
Overlaps end of result set.	sql_success	First row of rowset.	For rows in the rowset that overlap the result set, data is returned. For rows in the rowset outside the result set, the row status (rgfRowStatus) is SQL_ROW_NOROW and the contents of that part of the rowset buffers are undefined.
After end of result set.	sql_no_data_found	After end of rowset.	None. The contents of the rowset buffers are undefined.

The overlapping cases (second and fourth) are not symmetrical. For example, suppose a result set has 100 rows and the rowset size is 5. The following table shows the rowset and return code returned by SQLExtendedFetch for different values of **irow** when the fetch type is SQL_FETCH_RELATIVE (see below for the definition of this option).

Current Rowset	Irow	Return Code	New Rowset
1 to 5	-5	SQL_NO_DATA_FOUND	None
1 to 5	-3	SQL_SUCCESS	1 to 5
96 to 100	5	SQL_NO_DATA_FOUND	None
96 to 100	3	SQL_SUCCESS	99 and 100. For rows 3, 4, and 5 in the rowset, the corresponding row statuses are all set to SQL_ROW_NOROW.

fFetchType Argument

The **fFetchType Argument** is used to indicate the type that decides the position of the window (on the result set) for the rowset.

©Copyright 1995-2003 CASEMaker Inc.

The valid values for **fFetchType** are:

- SQL_FETCH_FIRST
- SQL_FETCH_LAST
- SQL_FETCH_NEXT
- SQL_FETCH_PRIOR
- SQL_FETCH_ABSOLUTE
- SQL_FETCH_RELATIVE
- SQL_FETCH_BOOKMARK

The **irow** argument is applied when the values SQL_FETCH_ABSOLUTE or SQL_FETCH_RELATIVE are used for the **fFetchType** argument. The rowset returned for SQL_FETCH_FIRST, SQL_FETCH_LAST, and SQL_FETCH_ABSOLUTE is not dependant on the value of fFetchType from the immediately previous SQLExtendedFetch call since it is not fetched relative to the current rowset.

The other values for **fFetchType** argument fetch a rowset according to the previous rowset:

- *SQL_FETCH_FIRST*: fetches the first rowset in the result set.
- *SQL_FETCH_LAST*: fetches the last complete rowset in the result set.
- *SQL_FETCH_ABSOLUTE*: fetches the rowset starting at row **irow** of the result set.

  If **irow** > 0, fetch the rowset starting at row **irow**.

  If **irow** < 0, fetch the rowset starting at row **irow**+result set size+1.

  E.g. if **irow** = -1 then the starting row of the returned rowset is the last row of the result set. If **irow** is less than 0, you can count back from the end of the result set to find the first row of the returned rowset.

  If **irow** = 0, return SQL_NO_DATA_FOUND and position the cursor before the start of result set. (To reset)

©Copyright 1995-2003 CASEMaker Inc.

- *SQL_FETCH_NEXT*: fetches the next rowset. If the cursor is currently positioned before the start of the result set (e.g. the initial condition), then this is equivalent to *SQL_FETCH_FIRST*.
- *SQL_FETCH_PRIOR*: fetches the previous rowset. If the cursor is currently positioned after the end of the result set, this is equivalent to SQL_FETCH_LAST.
- *SQL_FETCH_RELATIVE*: fetches the rowset starting at **irow** row from the start of the current rowset. If the cursor is positioned before the start of the result set:

  irow > 0: fetch the rowset starting at row **irow**. This is equivalent to the SQL_FETCH_ABSOLUTE value.

  irow < 0: return SQL_NO_DATA_FOUND without changing the cursor.

  If the cursor is positioned after the end of the result set:

  irow < 0: fetch the rowset starting at row **irow**+result set size+1. This is equivalent to SQL_FETCH_ABSOLUTE value.

  irow > 0: return SQL_NO_DATA_FOUND without changing the cursor.

  irow = 0: refresh (refetch) the current rowset.
- *SQL_FETCH_BOOKMARK*: fetches the rowset starting at the bookmark specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute.

irow Argument

The **irow** argument specifies the number of the row to fetch. You only need to use **irow** if the **fFetchType** argument is set to either SQL_FETCH_ABSOLUTE or SQL_FETCH_RELATIVE. Otherwise, you can ignore this value.

pcrow Argument

The **pcrow** argument specifies the number of rows (in the rowset buffer) actually fetched from an **SQLExtendedFetch** call. The range of valid values for **pcrow** is from 0 to the rowset size.

©Copyright 1995-2003 CASEMaker Inc.

rgfRowStatus Argument

The **rgfRowStatus** argument is an array of status values for all rows in the rowset. This array is allocated by an ODBC application.

The possible status values set by SQLExtendedFetch are:

- *SQL_ROW_NOROW:* data in this row is undefined.
- *SQL_ROW_SUCCESS:* data in this row was successfully fetched using SQLExtendedFetch.
- *SQL_ROW_ERROR:* an error was found when fetching this row using SQL*ExtendedFetch*.

For example, if the rowset size is ten and only nine rows are fetched by SQLExtendedFetch (e.g., with **fFetchType** set to SQL_FETCH_ABSOLUTE and **irow** set to -9), then the status value for the last row will be SQL_ROW_NOROW, while the status values for the other rows will be SQL_ROW_SUCCESS.

The SQLSetPos, which is used to manipulate the rows in a rowset fetched by SQLExtendedFetch, can only set the following values:

- *SQL_ROW_UPDATED:* the row is updated.
- *SQL_ROW_DELETED:* the row is deleted.
- SQL_ROW_ADDED: the row is added.

Returning Values and Processing Errors

SQLExtendedFetch fetches more than one row at a time. Each row fetched successfully or with a warning is given the value SQL_ROW_SUCCESS as the row status value. If an error is encountered, the row being fetched will be given the value SQL_ROW_ERROR and the fetch will stop. Subsequent rows in the rowset will be marked SQL_ROW_NOROW. The following 4 examples are possible, using a rowset size of 5.

➲ **Example 1**

All rows for rowset are fetched:

©Copyright 1995-2003 CASEMaker Inc.

```
     +------+-----------------+
row1 | data | SQL ROW SUCCESS |
     +------+-----------------+
row2 | data | SQL_ROW_SUCCESS |
     +------+-----------------+
row3 | data | SQL ROW SUCCESS |
     +------+-----------------+
row4 | data | SQL_ROW_SUCCESS |
     +------+-----------------+
row5 | data | SQL ROW SUCCESS |
     +------+-----------------+
```

➲ **Example 2**

All rows for rowset are fetched near the end of result set:

```
     +------+-----------------+
row1 | data | SQL_ROW_SUCCESS |
     +------+-----------------+
row2 | data | SQL_ROW_SUCCESS |
     +------+-----------------+
row3 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row4 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row5 | xxxx | SQL_ROW_NOROW   |
```

➲ **+------+----------+**

©Copyright 1995-2003 CASEMaker Inc.

➲ **Example 3**

Error found in fetching 2nd row:

```
     +------+-----------------+
row1 | data | SQL ROW SUCCESS |
     +------+-----------------+
row2 | xxxx | SQL_ROW_ERROR   |
     +------+-----------------+
row3 | xxxx | SQL ROW NOROW   |
     +------+-----------------+
row4 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row5 | xxxx | SQL ROW NOROW   |
     +------+-----------------+
```

➲ **Example 4**

No row fetched at all:

```
     +------+-----------------+
row1 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row2 | xxxx | SQL ROW NOROW   |
     +------+-----------------+
row3 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
row4 | xxxx | SQL ROW NOROW   |
     +------+-----------------+
row5 | xxxx | SQL_ROW_NOROW   |
     +------+-----------------+
```

©Copyright 1995-2003 CASEMaker Inc.

The return value of SQLExtendedFetch depends on the values of all the rows in the rowset. You should check the column status array for the status of the columns in each row.

The following return values are possible for SQLExtendedFetch:

- *SQL_SUCCESS*- if no errors or warnings were found and there was at least one row fetched.
- *SQL_NO_DATA_FOUND*- if there were no rows to fetch.
- *SQL_SUCCESS_WITH_INFO*- if there were any warnings found but no errors were found. If **SQLError** is called for warning information, the details of the last warning will be returned.
- *SQL_ERROR*- if an error was found. All subsequent **SQLExtendedFetch** calls will return the same error, and the ODBC application cannot access that result set any more. (E.g., a lock time-out is seen as an error even if the lock is later released. SQL*Execute* is needed to regenerate the result set.)

Table Modification Using SQLSetPos

If the result set is generated from a single table, and each row of the result set can be uniquely mapped to one row of the target table. Then each row of the rowset, which is a subset of result set, is associated with one physical row of the target table through an OID (object ID).

©Copyright 1995-2003 CASEMaker Inc.

➲ **Example 1**

Rowsets from the following query statements are all updateable:

```
create table t1 (c1 int, c2 int, c3 char(5))
     select * from t1;
     select * from t1 where c1 > 10;
     select c1 from t1 where c2 < 20;
     select c2, c1 from t1;
```

➲ **Example 2**

Rowsets from the followings query statement are not updateable:

```
create table t1 (c1 int, c2 int, c3 char(5))
     select * from t1,t2 ;
     select c1+c2 from t1 ;
     select c1*2 from t1 ;
```

The SQLSetPos now only supports modification of simple scans on a single table. Only expressions like **c1** can be modified. Expressions like **c1*2**, **c1+1**, and **c1+c2** are not modifiable.

If needed, the column default values are applied for those columns not in the projection, e.g. row insertion through SQLSetPos. Bound columns are a subset of the projection, which is inversely a subset of the table schema in a simple scan on a single table. You can only use SQLPutData to manipulate unbound columns.

Arguments of SQLSetPos

➲ **Prototype**

SQLSetPos:

```
RETCODE SQLSetPos(
                  HSTMT    hstmt,
                  UWORD    irow,
                  UWORD    fOption,
                  UWORD    fLock)
```

©Copyright 1995-2003 CASEMaker Inc.

The possible returned values from SQLSetPos are:

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NEED_DATA
- SQL_ERROR
- SQL_INVALID_HANDLE

Since SQLSetPos manipulates the rowset, you must call it after calling SQLExtendedFetch. The rowset to be operated on is from the previous SQLExtendedFetch. Note that the **rgfRowStatus** argument (the row status array) is set by SQLSetPos (according to different options), which is implicitly passed from the corresponding SQLExtendedFetch. Again, ODBC applications should check the SQLSetPos return value and row status array before SQLSetPos accesses the row data. If the result set is not modifiable, SQLSetPos will just return an error.

The possible values for row status set by SQLSetPos with different options are:

- SQL_ROW_SUCCESS
- SQL_ROW_ERROR
- SQL_ROW_NOROW
- SQL_ROW_UPDATED
- SQL_ROW_DELETED
- SQL_ROW_ADDED

irow Argument

irow is the number of the row in the rowset that the operation specified with **fOption** will be performed on. If the value for **irow** is 0, the operation will be applied to all rows of the rowset.

©Copyright 1995-2003 CASEMaker Inc.

Option Argument

The operation to apply on the rowset obtained from SQLExtendedFetch. Valid values are:

- SQL_POSITION
- SQL_REFRESH
- SQL_UPDATE
- SQL_DELETE
- SQL_ADD

SQL_POSITION positions the cursor on the rows of the rowset for those operations needing a cursor. This option does not change the row status array at all.

If **irow** = 0: position the cursor on the whole rowset.

If **irow** = n: position the cursor on row “n” (n =1 or <= rowset size).

If this option is used to position the cursor on more than one row, then the positioned statement is only performed on the first row of the selected rows.

➲ **Example**

```
/* hstmtS is for SQLExtendedFetch, SQLSetPos and SQLSetCursorName */
/* hstmtU is for positioned statement                              */
/* use hstmtS to query */

rc=SQLSetStmtOption(hstmtS, SQL ROWSET SIZE, ROWS);
rc=SQLSetCursorName(hstmtS, (UCHAR *)"C1", SQL NTS);
rc=SQLExecDirect(hstmtS, (UCHAR *)"SELECT NAME, BIRTHDAY FROM EMPLOYEE
                                   FOR UPDATE OF BIRTHDAY", SQL_NTS);
rc=SQLBindCol(hstmtS, 1, SQL C CHAR, szName, NAME LEN, cbName);
rc=SQLBindCol(hstmtS, 2, SQL_C_CHAR, szBirthday, BDAY_LEN, cbBirthday);

/* use hstmtS (through SQLExtendedFetch) to browse all rows */
```

©Copyright 1995-2003 CASEMaker Inc.

```
while (1) {
    rc=SQLExtendedFetch(hstmtS, SQL_FETCH_NEXT, 0, &crow,
                        rgfRowStatus);
    if (rc == SQL_ERROR || rc == SQL_NO_DATA_FOUND)
        break;
    for (irow = 0; irow < crow; irow++) {
        if (rgfRowStatus[irow] != SQL_ROW_DELETED)
              printf("%d %10s : %30s\n", irow+1, szName[irow],
szBirthday[irow]);
    }/*for*/
    /* read user input for line number and data to update */
    /* use hstmtS to position cursor for hstmtU           */
    /* use hstmtU to execute positioned update statement  */
    while (TRUE) {
      printf("\nRow number to update? (0 to quit)");
      gets((char *)szReply);
      irow = atoi((char *)szReply);
      if (irow > 0 && irow <= crow) {
          printf("\nNew birthday? ");
          gets((char *)szBirthday[irow-1]);
          rc=SQLSetPos(hstmtS, irow, SQL_POSITION, SQL_LOCK_NO_CHANGE);
          rc=SQLPrepare(hstmtU,
               (UCHAR *)"UPDATE EMPLOYEE SET BIRTHDAY=? WHERE CURRENT OF C1",
SQL_NTS);
          rc=SQLBindParameter(hstmtU, 1, SQL_PARAM_INPUT,
                              SQL_C_CHAR, SQL_CHAR, BDAY_LEN, 0,
                              szBirthday[irow-1], 0, NULL);
          rc=SQLExecute(hstmtU);
          rc=SQLFreeStmt(hstmtU,SQL_CLOSE);
      } else if (irow == 0) {
          break;
```


©Copyright 1995-2003 CASEMaker Inc.

```
        }
    }/*wh*/
}/*wh*/
```

SQL_REFRESH refreshes the row data of the rowset. This will refetch the same window into the rowset buffer. The row status of the newly fetched row is set to SQL_ROW_SUCCESS and the rows not in the result set are set to SQL_ROW_NOROW.

If **irow** = 0, position the cursor on the whole rowset and refresh.

If **irow** = n, DBMaker does not support other values for **irow** with SQL_REFRESH.

The row status for refreshed rows will be set to SQL_ROW_SUCCESS if this option is successful. The window (rowset) might be moved forward or backward in the result set if the refreshed rowset is full of “holes” created by the SQL_DELETE option.

SQL_UPDATE updates row data. The corresponding rows in the target table are updated with row data from the rowset buffer. The row status of the successfully updated rows is set to SQL_ROW_UPDATED. You cannot update a row that is marked SQL_ROW_DELETED.

If **irow** = 0, the cursor is positioned on the whole rowset and it is updated.

If **irow** = n, the cursor is positioned on row “n” and it is updated.

SQL_DELETE deletes the corresponding rows in the target table mapped by the rowset. You cannot delete a row that is marked SQL_ROW_DELETED. If rows are deleted (set as SQL_ROW_DELETED), you cannot perform the following operations on them: positioned UPDATE/DELETE statements, calls to SQLGetData, or calls to SQLSetPos with any options other than SQL_POSITION (you can only call SQLSetPos with SQL_SET_POSITION on rows set to SQL_ROW_DELETED).

If **irow** = 0, the cursor is positioned on the whole rowset and it is deleted.

If **irow** = n, the cursor is positioned on row “n” and it is deleted.

©Copyright 1995-2003 CASEMaker Inc.

SQL_ADD adds row data. The row status of the added rows is set to SQL_ROW_ADDED. If this option is applied, the rowset is just used as a user input buffer for data to be inserted. No corresponding rows in the target table are mapped by rows in the rowset. This is the only option that allows **irow** to be greater than the rowset size. This option does not change the cursor position (no cursor positioning is done). When inserting columns not bound to the rowset buffer, default values (if available) are used or NULL values (if default values are not available) are used.

If **irow** = 0,: add all rows of the rowset.

If **irow** = 1 ~ rowset size: add row **irow**.

If **irow** > *rowset* size: row **irow** is still found from the start of rowset buffer with an appropriate offset. For example, if **irow** = rowset size + 1, or **irow** = rowset size + 2, add row **irow**.

An ODBC application allocates more buffer space than the rowset buffer space, specified by SQL_ROWSET_SIZE. This simplifies ODBC application programming. If no extra buffer is allocated and **irow** is greater than the rowset size, then an application program error may result from trying to access illegal memory addresses.

fLock Argument

To lock or unlock the corresponding operated rows in target table.

The valid value for fLock is:

- *SQL_LOCK_NO_CHANGE:* do not change the row's lock mode.

Column Indicators

When an ODBC application wants to insert a NULL value into a column, the only interface is the column indicator (status). There is no host variable (like the host variables from INSERT INTO t1 VALUES (?,?)) that we can retrieve information from or specify that allows insertion of NULL values into columns. With SQLExtendedFetch and SQLSetPos, the column indicators from SQLBindCol are the only interface for specifying information for both fetching and modification.

©Copyright 1995-2003 CASEMaker Inc.

For example, if an ODBC application wants to update a column with NULL values (or insert a NULL value to that column), then the corresponding column indicator should be set to SQL_NULL_DATA before calling SQLSetPos with the SQL_UPDATE or SQL_INSERT options. The same method is used if the default value is to be applied, but the column indicator should be set to SQL_DEFAULT_PARAM.

SQLPutData

SQLGetData, if used with SQLExtendedFetch, is the only way to fetch unbound columns (which are mostly BLOB/file object columns). Similarly, you call SQLSetPos with SQLPutData to modify those unbound columns (which are mostly BLOB/file object columns). There is no column indicator to use and you can only use the argument **cbValue** of SQLPutData. The rowset size must be 1 before executing SQLGetData or SQLPutData.

For non-projection columns, default values are applied by SQLSetPos with SQL_ADD options.

➲ **Example 1**

Create table:

```
create table t1 (c1 int, c2 int, c3 char(5) default 'col3')
```

➲ **Example 2**

A Select query:

```
select c1, c2 from t1
```

➲ **Example 3**

Modify table t1using the following calls:

```
/* bind columns c1, c2 , execute and fetch */

SQLBindCol(hstmt, 1, SQL C CHAR, c1 rgbValue, c1 len ,c1 pcbValue);

SQLBindCol(hstmt, 2, SQL_C_CHAR, c2_rgbValue, c2_len ,c2_pcbValue);

SQLExecute(hstmt);

SQLExtendedFetch(hstmt, SQL_FETCH_FIRST, 0, &crow, rgfRowStatus);
```

©Copyright 1995-2003 CASEMaker Inc.

```
/* specify c1, c2 values in first row of rowset  */
/* first row of rowset is used as input buffer   */
SQLSetPos(hstmt, 1, SQL_ADD, SQL_LOCK_NO_CHANGE);  /* default for c3 */
/* specify c1,c2 values in first row of rowset   */
/* first row of rowset is used as input buffer   */
/* update the row (in table t1) corresponding to */
/* first row in rowset */
SQLSetPos(hstmt, 1, SQL_UPDATE, SQL_LOCK_NO_CHANGE);/* c3 is not changed */
```

Column **c3** is not in the projection and only column **c1** and **c2** can be found in the rowset. In other words, to insert one extra tuple, SQLSetPos gets values from the rowset for columns **c1** and **c2** and uses the default value 'col3' for column **c3**. To update the corresponding row in **t1** for the first row in the *rowset*, be certain column **c3** has not been changed since **c3** is not in the projection.

For bound columns, SQLSetPos gets all the input data it needs (for the SQL_ADD and SQL_UPDATE options) from the rowset (the bound buffer).

For each unbound column, if it is neither a BLOB (LONG VARCHAR or LONG VARBINARY) nor a file object type, then the default value is still applied if needed for SQLSetPos. You cannot use SQLPutData to put data for this type of column since the default value is used when SQLSetPos is executed.

For unbound BLOB/file object columns, we must use SQLPutData to modify them. Before SQLPutData and after SQLSetPos, we still need to execute SQLParamData to find all the unbound BLOB/file object columns.

BLOB(LONG VARCHAR and LONG VARBINARY) columns:

- To input *NULL* values; call SQL*PutData* with argument **cbValue** set to *SQL_NULL_DATA.*
- To input default values, call SQL*PutData* with argument **cbValue** set to *SQL_DEFAULT_PARAM.*
- To input data; call SQL*PutData* with input data in argument **rgbValue** and *SQL_NTS* or the length of **rgbValue** in argument **cbValue**. To input data the

©Copyright 1995-2003 CASEMaker Inc.

SQL_C_TYPE for the *LONG VARCHAR* and *LONG VARBINARY* data types are *SQL_C_LONGVARCHAR* and *SQL_C_LONGVARBINARY*.

File object columns:

- To input *NULL* values, call SQL*PutData* with argument **cbValue** *set* to *SQL_NULL_DATA.*
- To input default values, call SQL*PutData* with argument **cbValue** set to *SQL_DEFAULT_PARAM*.

➲ **Example 4**

Execute SQLSetPos, make the call, and execute SQLPutData to input data:

```
SQLSetStmtOption(hstmt, SQL_SPOS_FO, SQL_SPOS_FO_DATA|col);
```

This specifies the proceeding SQLPutData call to insert data from argument rgbValue into column **col**, which is a file object column in the projection. The **col** is the index of the object column in the target file of the projection. The option SQL_SPOS_FO_DATA will force the system to use SQL_C_CHAR or SQL_LONGVARCHAR to bind the input data. A system file will be automatically created for this type of data input.

➲ **Example 5**

Execute SQLSetPos, make the call, and execute SQLPutData to input user files:

```
SQLSetStmtOption(hstmt, SQL_SPOS_FO, SQL_SPOS_FO_SFILE|col)
```

This specifies the proceeding SQLPutData call to insert a user file with a filename specified in argument **rgbValue** into column **col**, which is a file object column in the projection. The option SQL_SPOS_FO_SFILE will force the system to use SQL_C_CHAR or SQL_FILE to bind the input data.

➲ **Example 6**

Execute SQLSetPos, make the call, and execute SQLPutData to input system files:

```
SQLSetStmtOption(hstmt, SQL_SPOS_FO, SQL_SPOS_FO_CFILE|col)
```

This specifies the proceeding **SQLPutData** call to insert a system file with a filename specified in argument **rgbValue** into column **col**, which is a file object column in the

©Copyright 1995-2003 CASEMaker Inc.

projection. The option **SQL_SPOS_FO_CFILE** will force the system to use **SQL_C_FILE** or **SQL_LONGVARCHAR** to bind the input data.

The following shows how to use **SQLSetPos** and **SQLPutData** to input BLOB and file object data:

➲ **Example 7**

Create table schema:

```
create table t1 (c1 int, c2 long varchar, c3 file, c4 int default 10)
```

➲ **Example 8**

A Select query:

```
select c2, c3, c4 from t1
```

➲ **Example 9**

Using code:

```
/* do not bind any column                                                  */
SQLSetStmtOption(hstmt, SQL ROWSET SIZE, 1);
       :
/* execute and fetch                                                       */
SQLExecute(hstmt);
SQLExtendedFetch(hstmt, SQL FETCH FIRST, 0, &crow, rgfRowStatus);
/* call SQLSetPos to insert one tuple                                      */
/* SQLSetPos returns SQL_NEED_DATA                                         */
SQLSetPos(hstmt, 1, SQL ADD, SQL LOCK NO CHANGE);  /* default(10) for c4 */
/* input null for c2(long varchar)                                         */
SQLParamData(hstmt, (void *)&paranum);
SQLPutData(hstmt, buf,SQL_NULL_DATA);
/* input user file for c3(file)                                            */
SQLParamData(hstmt, (void *)&paranum);
/* specify user file input and place file name in sbuf                     */
SQLSetStmtOption(hstmt, SQL_SPOS_FO, SQL_SPOS_FO_SFILE|2);   /* 2 for c3 */
```


©Copyright 1995-2003 CASEMaker Inc.

```
        :
/* input user file for c3(file)                                          */
SQLPutData(hstmt, sbuf, strlen(sbuf));
```

Using SQLSetPos

SQLSetPos modifies more than one row at a time, so the rules for the return value are similar to SQLExtendedFetch. Each successful (or with warning) operation on a row is marked according to the option used. The row will be marked SQL_ROW_ERROR if an error is found and operation is not stopped unless it is a critical error such as an aborted transaction.

The rules are:

- Return *SQL_SUCCESS* if no errors or warnings are found.
- Return *SQL_NO_DATA_FOUND* if there are no rows to fetch (for the *SQL_REFRESH* option only).
- Return *SQL_SUCCESS_WITH_INFO* if there are any warnings or errors found during the operation. We can call *SQLError* to get complete error information. If there are only warnings and no errors, then only the last warning is recorded.
- Return *SQL_ERROR* if a critical error is found in operation.

Note that the partial result of SQLSetPos done before the tuple where the error is found is not undone. That is, this function is not an atomic operation. Each call to SQLSetPos (except with the options SQL_REFRESH and SQL_POSITION) will commit work after it is successfully executed if autocommit mode is on.

LIMITATIONS OF SQLSETPOS

Result sets from a query with a subquery are not modifiable; you cannot call SQLSetPos for that type of result set.

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.

6 Error Handling

After reading the previous chapters, you should be able to construct an ODBC program. However, what do you do when problems occur while calling ODBC functions? In this chapter, you will learn how to get error information when an error occurs. This chapter also introduces some ODBC catalog functions that allow you to get information from the system catalog (system tables). Some other ODBC functions are also covered here, including functions that are used to get system information about the data source, such as supported data types, supported built-in functions, and supported ODBC functions.

In this chapter you will learn how to:

- Get detailed error information when a call to an *ODBC* function fails by using the *SQLError* function.
- Retrieve catalog information such as table schemas and statistics information by using catalog functions such as *SQLTables*, *SQLColumns*, *SQLStatistics,* and *SQLSpecialColumns*.
- Obtain system information about the data source by using the *SQLGetTypeInfo*, *SQLGetInfo*, and *SQLGetFunctions* functions.

NOTE *Error information is collected differently using* DBMaker *(ODBC 3.0) from what is described in this chapter. For more information, refer to chapter 8 on* "DBMaker 3.0 Functions".

©Copyright 1995-2003 CASEMaker Inc.

6.1 Retrieving Error Information

When an application executes an ODBC function and an error code is returned, it needs detailed error information to determine what caused the error. This section explains how to use the SQLError function to retrieve error information.

Common Error Codes Defined in ODBC

After calling an ODBC function, you may get one of the following return codes:

- ***SQL_SUCCESS*** — the ODBC function executed successfully.
- ***SQL_SUCCESS_WITH_INFO*** — the ODBC function was executed successfully, but some warning information is being returned.
- ***SQL_NO_DATA_FOUND*** — no more data can be fetched.
- ***SQL_ERROR*** — an error has occurred and the function failed.
- ***SQL_INVALID_HANDLE*** — an invalid handle was detected and the function failed.
- ***SQL_NEED_DATA*** — the driver indicates the application must send parameter data values.

If an application calls any ODBC function (except SQLError itself) and the return code is **SQL_ERROR** or **SQL_SUCCESS_WITH_INFO**, it can call SQLError to get additional error information.

How to Use SQLError

SQLError is used to get error information in the input handle, including the error message, error state, and the driver's native error code. The driver's native error code is the error code defined by each driver. This may be different for different drivers. (For native DBMaker error codes, see *Appendix C*.)

Applications call SQLError when the error code returned by the previous ODBC function is **SQL_ERROR** or **SQL_SUCCESS_WITH_INFO**.

©Copyright 1995-2003 CASEMaker Inc.

➲ **Prototype**

SQLError:

```
RETCODE  SQLError(
                  HENV        henv,
                  HDBC        hdbc,
                  HSTMT       hstmt,
                  UCHAR  FAR *szSqlState,
                  SDWORD FAR *pfNativeError,
                  UCHAR  FAR *szErrorMsg,
                  SWORD       cbErrorMsgMax,
                  SWORD  FAR *pcbErrorMsg)
```

The three handles in the argument list of SQLError are not all necessarily passed to SQLError. The ODBC driver will find the associated return code from the rightmost non-null handle.

➲ **Example 1**

```
SQLError(henv, hdbc, hstmt, ....)
```

➲ **Example 2**

Returned error information is in **hstmt**:

```
SQLError(SQL_NULL_ENV, hdbc, SQL_NULL_STMT, ...)
```

The driver will return the error information associated with hdbc. You should ensure the applications pass the proper handles to SQLError so that the error information they need can be retrieved successfully.

If there is no error information to be retrieved, SQLError will return **SQL_NO_DATA_FOUND**. Each time after SQLError is called and error information is returned, the error information in that handle will be cleared. This means the error information for one ODBC function call can be retrieved only once.

The SQL Access Group SQL CAE specification (1992) and X/Open define **SQLSTATE** values returned by SQLError. The values are 5-character strings with a two-character class value followed by a three-character subclass value. For example, the

©Copyright 1995-2003 CASEMaker Inc.

class value 01 is a warning and the corresponding return code is **SQL_SUCCESS_WITH_INFO**. Refer to the "*Microsoft ODBC Programmer's Reference*", for more detailed information regarding **SQLSTATE** values defined in ODBC.

Example

SQLError with SQLState:

```
#define  MSG_LEN  256       /* error message buffer length          */
HENV   henv;                /* environment handle                   */
HDBC   hdbc;                /* connection handle                    */
HSTMT  hstmt;               /* statement handle                     */
SDWORD retcode, retcode1;   /* return code                          */
UCHAR  sqlState[6];         /* buffer to store SQLSTATE             */
SDWORD nativeErr;           /* native error code                    */
UCHAR  errMsg[MSG LEN];     /* buffer to store error message        */
SWORD  realMsgLen;          /* real length of returned error message */
retcode = SQLAllocEnv(&henv);
retcode = SQLAllocConnect(&hdbc);
/* Use specified DB_NAME(data source name), uid (user id),              */
/* pwd (password) to connect to a data source.  If any warnings or      */
/* errors are detected, call SQLError and pass hdbc to retrieve         */
/* error information from the connection handle with other handles      */
/* set to NULL.  Then print the error information and return.           */
retcode = SQLConnect(hdbc, DB_NAME, SQL_NTS, uid, SQL_NTS, pwd,
                          SQL_NTS);
if (retcode != SQL SUCCESS)    /* warning or error returned         */
    {
     retcode1 = SQLError(SQL_NULL_HDBC, hdbc, SQL_NULL_HSTMT, sqlState,
                         &nativeErr, errMsg, MSG_LEN, &realMsgLen);
     print err(sqlState, nativeErr, errMsg, realMsgLen);
     return;
```


©Copyright 1995-2003 CASEMaker Inc.

```
    }
/* Get SQL command string and execute it.  If any warnings or errors     */
/* are detected, call SQLError and pass hstmt to retrieve error          */
/* information from the statement handle, then print the error           */
/* information and return.                                               */

retcode = execute_cmd(hstmt);   /* execute a SQL command            */
if (retcode != SQL_SUCCESS)     /* warning or error returned        */
    {
     retcode1 = SQLError(SQL_NULL_HDBC, SQL_NULL_HDBC, hstmt, sqlState,
                         &nativeErr, errMsg, MSG_LEN, &realMsgLen);
     print_err(sqlState, nativeErr, errMsg, realMsgLen);
     return;
    }
```

Error Queues

ODBC allows multiple error codes to be stored in an error queue and are associated with one handle. SQLError can be called multiple times to retrieve error codes one by one. Currently DBMaker will only return multiple errors for *Database Consistency Checking* (DBCC) operations. These errors are stored in the error queue.

An application can call SQLError many times until all the errors in the error queue are fetched. Once all errors have been fetched, SQLError will return **SQL_NO_DATA_FOUND**.

➲ **Example**

An application checking the consistency of an account table:

```
#define  MSG_LEN  256       /* error message buffer length           */
UCHAR   sqlState[6];        /* buffer to store SQLSTATE              */
SDWORD  nativeErr;          /* native error code                     */
UCHAR   errMsg[MSG LEN];    /* buffer to store error message         */
SWORD   realMsgLen;         /* real length of returned error message */
```

©Copyright 1995-2003 CASEMaker Inc.

```
SWORD   count;
SWORD   retcode;
retcode = SQLExecute(hstmt, "check table account", SQL_NTS);

do {
    retcode = SQLError(SQL_NULL_HDBC, hdbc, SQL_NULL_HSTMT, sqlState,
                       &nativeErr, errMsg, MSG_LEN, &realMsgLen);
    if (retcode == SQL_NO_DATA_FOUND)
        {
         printf("check error queue finish \n\n");
         break;
         }
    count++;
    printf("-->Error %d :\n", count);
    printf("   SQLSTATE = %s \n", sqlState);
    printf("   native error = %ld \n", nativeErr);
    printf("   error message = %s \n", errMsg);
    printf("   error message length = %d \n", realMsgLen);
   }
while ((retcode == SQL_SUCCESS) || (retcode == SQL_SUCCESS_WITH_INFO));
```


©Copyright 1995-2003 CASEMaker Inc.

6.2 Catalog Functions

There are several system tables in a relational database which record information about tables, columns, and privileges, etc., known as *catalogs*. The catalogs can be used to read the schema information for tables and indexes in a database.

All of the catalog functions work in the same manner. Specify information using parameters when calling the catalog functions, and a result set will be returned. You can then fetch data from the result set.

In this section, four commonly used catalog functions: SQLTables, SQLColumns, SQLStatistics, and SQLSpecialColumns will be introduced.

- ***SQLTables*** - gets a list of table or view names in a database.
- ***SQLColumns*** - gets column information about specified tables.
- ***SQLStatistics*** - gets statistics information about tables and the indexes associated with those tables.
- ***SQLSpecialColumns*** - gets the optimal set of columns that uniquely identifies a row in a table.

Search Patterns

Some arguments in the catalog functions accept search patterns to select the desired object. The simplest search pattern is a character string that is used to match exactly the item you are looking for. In addition, you can use wild card metacharacters in a search pattern to do searches that are more powerful. DBMaker supports the following metacharacters: underscore (_), percent (%) and the escape character (\).

- The underscore character *(_)* is used to match any one character.
- The percent character *(%)* is used to match zero or more characters.
- The escape character *(\)* permits the metacharacters *%* or _ to be used as literal characters in search patterns. To use the escape character \ as a literal character in search pattern, just include it twice *(\\)*.

©Copyright 1995-2003 CASEMaker Inc.

For example, if the search pattern for a table name is %A%, the function will return all tables with names that contain the character A. If the search pattern for a table name is _A_, the function will return all tables with names that are three characters long with A as the middle character. If the search pattern for a table name is %, the function will return all tables.

If you want to retrieve information for a table named ***TAB_TEST***, and you use ***TAB_TEST*** as the search pattern, then you will also get information for the tables ***TAB1TEST***, ***TAB2TEST***, etc. in the result set. This is not what you want. To solve this problem put an escape character in front of the metacharacter: ***TAB_TEST***.

NOTE *When passing this string through a C compiler, you must specify* TAB_TEST *instead of* TAB_TEST. *This is because the* C *compiler also treats* "\" *as an escape character. See the following example for* SQLTables.

SQLTables

When you connect to a database and want to determine information about all or a particular set of tables, calling SQLTables with the specified criteria will get the answer.

➲ **Prototype**

SQLTables:

```
RETCODE SQLTables (
                   HSTMT   hstmt,
                   UCHAR   *szTableQualifier,
                   SWORD   cbTableQualifier,
                   UCHAR   *szTableOwner,
                   SWORD   cbTableOwner,
                   UCHAR   *szTableName,
                   SWORD   cbTableName,
                   UCHAR   *szTableType,
                   SWORD   cbTableType)
```


©Copyright 1995-2003 CASEMaker Inc.

The arguments for SQLTables are:

- *hstmt* — a valid statement handle for retrieved results.
- *szTableQualifier* — not supported by *DBMaker*, it should be ***NULL*** or an empty string.
- *cbTableQualifier* — the length of *szTableQualifier*, it should be zero.
- *szTableOwner* — points to the owner name. The owner is the person who created the table or view. It can be a string to be used as a search pattern or a ***NULL*** value. Use a ***NULL*** value to indicate all owners.
- *cbTableOwner* — the length of *szTableOwner* or ***SQL_NTS***.
- *szTableName* — points to the names of tables or views. It can be a string to be used as a search pattern or a ***NULL*** value. Use a ***NULL*** value to indicate all names.
- *cbTableName* — the length of *szTableName* or ***SQL_NTS***.
- *szTableType* — the list of table types *(TABLE and/or VIEW)*.
- *cbTableType* — the length of *szTableType*.

NOTE *A string to be used as a search pattern can exist in* **szTableQualifier**, **szTableOwner** *and* **szTableName**. *These three arguments and their corresponding string length arguments,* **cbTableQualifier**, **cbTableOwner** *and* **cbTableName** *also appear in the other three catalog functions: SQLColumns, SQLStatistics and SQLSpecialColumns.*

SQLTables returns a result set consisting of the following columns:

Column No.	Column Name	Data Type
1	TABLE_QUALIFIER	VARCHAR(128)
2	TABLE_OWNER	VARCHAR(128)
3	TABLE_NAME	VARCHAR(128)
4	TABLE_TYPE	VARCHAR(128)
5	REMARKS	VARCHAR(254)

©Copyright 1995-2003 CASEMaker Inc.

SQLTables returns a result set according to the user's criteria. For example, when szTableName is %A% and szTableOwner is _A_, the result set will contain all tables whose names contain the character A and that have owners whose names are three characters long with A as the middle character. If you want to find the names of all tables in the database, you only need to set szTableQualifier, szTableOwner and szTableName to **NULL**.

In fact, you can regard SQLTables as a way to execute a query using SQLExecDirect. This means you need to use **SQLFetch** to get the result set. Before using SQLFetch, you should use SQLBindCol to bind the columns in the result set.

The following code gives an example for SQLTables. Suppose there are two tables named *TAB_TEST1* and *TAB_TEST2*. After calling SQLTables, you will get information for *TAB_TEST1* and *TAB_TEST2*, ordered by *TABLE_TYPE*, *TABLE_QUALIFIER*, *TABLE_OWNER*, and *TABLE_NAME*.

➲ **Example**

```
HDBC   hdbc;
HSTMT  hstmt;
UCHAR  tabQualifier[255], tabOwner[255], tabName[255],
UCHAR  tabType[255], remarks[255];
SDWORD lenTabQualifier, lenTabOwner, lenTableName;
SDWORD lenTableType, lenRemarks;
SDWORD retcode;
...
retcode = SQLAllocStmt(hdbc,&hstmt);
retcode = SQLTables(hstmt,
                    (UCHAR FAR *)NULL, 0,            /* tabQualifier  */
                    (UCHAR FAR *)NULL, 0,            /* tabOwners     */
                    (UCHAR FAR *)"DB\\_%", SQL_NTS,  /* table name    */
                    (UCHAR FAR *)"TABLE", SQL_NTS);  /* table type    */
/* Bind columns in result set to storage locations                      */

retcode = SQLBindCol(hstmt, 1, SQL_C_CHAR, tabQualifier, 255,
                     &lenTabQualifier);
```


©Copyright 1995-2003 CASEMaker Inc.

```
retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, tabOwner, 255, &lenTabOwner);
retcode = SQLBindCol(hstmt, 3, SQL C CHAR, tabName, 255, &lenTableName);
retcode = SQLBindCol(hstmt, 4, SQL C CHAR, tabType, 255, &lenTableType);
retcode = SQLBindCol(hstmt, 5, SQL_C_CHAR, remarks, 255, &lenremarks);
while ((retcode = SQLFetch(hstmt)) == SQL SUCCESS)
      {                    /* print out the record in the result set */
       printf("column 1 : table qualifier = %s\n", tabQualifier);
       printf("column 2 : table owner = %s\n", tabOwner);
       printf("column 3 : table name = %s\n", tabName);
       printf("column 4 : table type = %s\n", tabType);
       printf("column 5 : remarks = %s\n", remarks);
      }
...
```

NOTE *When a function returns a result set, the user should use SQLBindCol and SQLFetch to get the rows in the result set. SQLTables, SQLColumns, SQLStatistics, and SQLSpecialColumns are all such functions.*

SQLColumns

You can use SQLTables to get the names of tables that are in a database. Similarly, you can use the SQLColumns function to get information on the columns found in a particular table.

➲ **The prototype for SQLColumns is**

```
RETCODE SQLColumns (
                    HSTMT    hstmt,
                    UCHAR   *szTableQualifier,
                    SWORD    cbTableQualifier,
                    UCHAR   *szTableOwner,
                    SWORD    cbTableOwner,
                    UCHAR   *szTableName,
                    SWORD    cbTableName,
                    UCHAR   *szColumnName,
                    SWORD    cbColumnName);
```

©Copyright 1995-2003 CASEMaker Inc.

Where **szTableQualifier**, **cbTableQualifier**, **szTableOwner**, **cbTableOwner**, **szTableName**, and **cbTableName** are defined the same as **SQLTables**. **szColumnName** points to the search pattern string of the column name. **cbColumnName** is the length of **szColumnName**.

Like the **SQLTables** function, a result set that matches the criteria in the arguments given above is returned containing the column information.

The following table lists the columns of the result set:

Column No.	Column Name	Data Type	Comments
1	TABLE_QUALIFIER	VARCHAR(128)	
2	TABLE_OWNER	VARCHAR(128)	
3	TABLE_NAME	VARCHAR(128)	NOT NULL
4	COLUMN_NAME	VARCHAR(128)	NOT NULL
5	DATA_TYPE	SMALLINT	NOT NULL
6	TYPE_NAME	VARCHAR(128)	NOT NULL
7	PRECISION	INTEGER	
8	LENGTH	INTEGER	
9	SCALE	SMALLINT	
10	RADIX	SMALLINT	
11	NULLABLE	SMALLINT	NOT NULL
12	REMARKS	VARCHAR(254)	

The result set is ordered by: ***TABLE_QUALIFIER***, ***TABLE_OWNER,*** and ***TABLE_NAME***. You should use SQLBindCol to bind the columns in the result set and then use SQLFetch to fetch the results.

©Copyright 1995-2003 CASEMaker Inc.

SQLStatistics

SQLStatistics retrieves a list of statistics about specified table(s) and the indexes associated with those table(s).

➲ **Prototype**

SQLStatistics:

```
RETCODE SQLStatistics (
                        HSTMT    hstmt,
                        UCHAR   *szTableQualifier,
                        SWORD    cbTableQualifier,
                        UCHAR   *szTableOwner,
                        SWORD    cbTableOwner,
                        UCHAR   *szTableName,
                        SWORD    cbTableName,
                        UWORD    fUnique,
                        UWORD   fAccuracy)
```

Where **szTableQualifier**, **cbTableQualifier**, **szTableOwner**, **cbTableOwner**, **szTableName**, and **cbTableName** are defined the same as for **SQLTables** and **SQLColumns**. **fUnique** is used to specify the type of index to be returned and **fAccuracy** is used to specify the importance of the ***CARDINALITY*** and ***PAGES*** columns in the result set.

NOTE ***fUnique*** *has two options: SQL_INDEX_UNIQUE or SQL_INDEX_ALL.* ***fAccuracy*** *also has two options: SQL_ENSURE or SQL_QUICK.*

The following table lists the columns in the result set:

Column No.	Column Name	Data Type	Comments
1	TABLE_QUALIFIER	VARCHAR(128)	
2	TABLE_OWNER	VARCHAR(128)	
3	TABLE_NAME	VARCHAR(128)	NOT NULL
4	NON_UNIQUE	SMALLINT	

©Copyright 1995-2003 CASEMaker Inc.

Column No.	Column Name	Data Type	Comments
5	INDEX_QUALIFIER	VARCHAR(128)	
6	INDEX_NAME	VARCHAR(128)	
7	TYPE	SMALLINT	NOT NULL
8	SEQ_IN_INDEX	SMALLINT	
9	COLUMN_NAME	VARCHAR(128)	
10	COLLATION	CHAR(1)	
11	CARDINALITY	INTEGER	
12	PAGES	INTEGER	
13	FILTER_CONDITION	VARCHAR(128)	

The ***TYPE*** column either has the value **SQL_TABLE_STAT** or **SQL_INDEX_OTHER**. **SQL_TABLE_STAT** indicates the row contains statistics for a table and the ***NON_UNIQUE***, ***INDEX_QUALIFIER***, ***INDEX_NAME***, ***SEQ_IN_INDEX***, ***COLUMN_NAME***, ***COLLATION,*** and ***FILTER_CONDITION*** columns (used for indexes) will be **NULL**. On the other hand, **SQL_INDEX_OTHER** indicates the row contains statistics for an index.

Similar to SQLTables and SQLColumns, you need SQLBindCol and SQLFetch to retrieve the data in the result set. The order of the columns in the result set is ***NON_UNIQUE***, ***TYPE***, ***INDEX_QUALIFIER***, ***INDEX_NAME,*** and ***SEQ_IN_INDEX***. For a code example of a similar function, please refer to the SQLTables code example.

SQLSpecialColumns

As the function name implies, SQLSpecialColumns returns the special columns that uniquely specify rows in a table.

➲ **Prototype**

SQLSpecialColumns:

```
RETCODE SQLSpecialColumns (
                            HSTMT    hstmt,
                            UWORD    fColType,
                            UCHAR   *szTableQualifier,
```


©Copyright 1995-2003 CASEMaker Inc.

```
                    SWORD    cbTableQualifier,
                    UCHAR   *szTableOwner,
                    SWORD    cbTableOwner,
                    UCHAR   *szTableName,
                    SWORD    cbTableName,
                    UWORD    fScope,
                    UWORD    fNullable);
```

Where hstmt is a valid statement handle, and szTableQualifier, cbTableQualifier, szTableOwner, cbTableOwner, szTableName, cbTableName are all defined the same as for SQLTables. fColType specifies the type of column to return. fScope is the minimum required scope of the special column. fNullable determines whether to return special columns that can have a **NULL** value.

NOTE ***fColType*** *has two options: SQL_BEST_ROWID and SQL_ROWVER.* ***fScope*** *has three options: SQL_SCOPE_CURROW, SQL_SCOPE_TRANSACTION and SQL_SCOPE_SESSION.* ***fNullable*** *has two options: SQL_NO_NULLS and SQL_NULLABLE.*

The following table lists the columns in the result set:

Column No.	Column Name	Data Type	Comments
1	SCOPE	SMALLINT	
2	COLLUMN_NAME	VARCHAR(128)	NOT NULL
3	DATA_TYPE	SMALLINT	NOT NULL
4	TYPE_NAME	VARCHAR(128)	NOT NULL
5	PRECISION	INTEGER	
6	LENGTH	INTEGER	
7	SCALE	SMALLINT	
8	PSEUDO_COLUMN	SMALLINT	

DBMaker provides a specific row identifier, *OID*, which is similar to *ROWID* in Oracle or *TID* in Ingres. OID is treated as a pseudo-column in a table because a query like SELECT * FROM ACCOUNT will not return such a column name, but you can still use OID in a select list or WHERE clause to fetch the records you want by explicitly specifying it.

©Copyright 1995-2003 CASEMaker Inc.

Once you specify **SQL_BEST_ROWID** in fColType, the result set returned by SQLSpecialColumns simply contains a row whose column name is OID. You can use this special column to re-select that row within the defined scope in fScope. The result of the SELECT statement is guaranteed to have either no rows or one row. For a code example of a similar function, please reference SQLTables.

If the ***fColType***, ***fScope***, or ***fNullable*** arguments specify characteristics that are not supported by DBMaker, SQLSpecialColumns returns a rowset with no rows. A subsequent call to SQLFetch or SQLExtendedFetch on the ***hstmt*** will return ***SQL_NO_DATA_FOUND***.

©Copyright 1995-2003 CASEMaker Inc.

6.3 System Information

You can use SQLGetTypeInfo, SQLGetInfo, and SQLGetFunctions to get system information about the data source. These ODBC functions are illustrated by examples in the following sections.

SQLGetTypeInfo

You can use SQLGetTypeInfo to get information about data types supported by the data source.

➲ **Prototype**

SQLGetTypeInfo:

```
RETCODE  SQLGetTypeInfo (HSTMT hstmt, SWORD fSqlType)
```

When given a value for fSqlType, SQLGetTypeInfo returns the related type information in the result set. You can use SQLBindCol to bind output storage for the result set and use SQLFetch to fetch the results into the output storage. fSqlType can be any SQL data type — **SQL_CHAR**, **SQL_DECIMAL**, **SQL_INTEGER**, etc.

The result set is:

Column No.	Column Name	Data Type	Comments
1	TYPE_NAME	VARCHAR(128)	NOT NULL
2	DATA_TYPE	SMALLINT	NOT NULL
3	PRECISION	INTEGER	
4	LITERAL_PREFIX	VARCHAR(128)	
5	LITERAL_SUFFIX	VARCHAR(128)	
6	CREATE_PARAMS	VARCHAR(128)	
7	NULLABLE	SMALLINT	NOT NULL
8	CASE_SENSITIVE	SMALLINT	NOT NULL
9	SEARCHABLE	SMALLINT	NOT NULL
10	UNSIGNED_ATTRIBUTE	SMALLINT	
11	MONEY	SMALLINT	NOT NULL

©Copyright 1995-2003 CASEMaker Inc.

Column No.	Column Name	Data Type	Comments
12	AUTO_INCREMENT	SMALLINT	
13	LOCAL_TYPE_NAME	VARCHAR(128)	
14	MINIMUM_SCALE	SMALLINT	
15	MAXIMUM_SCALE	SMALLINT	

This following uses SQLGetTypeInfo with **SQL_ALL_TYPES** as the value of fSqlType to fetch all data types supported by the data source.

Example

```
UCHAR  name[30], prefix[30], suffix[30], params[30], local_name[30];
SWORD  type, nullable, case_sen, searchable, unsign, money, auto_inc;
SWORD  min scale, max scale;
UDWORD prec;
SDWORD len[15], retcode;

/* bind all columns                                                    */
retcode = SQLBindCol(hstmt,  1, SQL_C_CHAR,  name,       30, &len[1]);
retcode = SQLBindCol(hstmt,  2, SQL_C_SHORT, &type,       0, &len[2]);
retcode = SQLBindCol(hstmt,  3, SQL_C_LONG,  &prec,       0, &len[3]);
retcode = SQLBindCol(hstmt,  4, SQL C CHAR,  prefix,     30, &len[4]);
retcode = SQLBindCol(hstmt,  5, SQL_C_CHAR,  suffix,     30, &len[5]);
retcode = SQLBindCol(hstmt,  6, SQL_C_CHAR,  params,     30, &len[6]);
retcode = SQLBindCol(hstmt,  7, SQL_C_SHORT, &nullable,   0, &len[7]);
retcode = SQLBindCol(hstmt,  8, SQL C SHORT, &case sen,   0, &len[8]);
retcode = SQLBindCol(hstmt,  9, SQL_C_SHORT, &searchable, 0, &len[9]);
retcode = SQLBindCol(hstmt, 10, SQL_C_SHORT, &unsign,     0, &len[10]);
retcode = SQLBindCol(hstmt, 11, SQL_C_SHORT, &money,      0, &len[11]);
retcode = SQLBindCol(hstmt, 12, SQL C SHORT, &auto inc,   0, &len[12]);
retcode = SQLBindCol(hstmt, 13, SQL_C_CHAR,  local_name, 30, &len[13]);
retcode = SQLBindCol(hstmt, 14, SQL_C_SHORT, &min_scale,  0, &len[14]);
```


©Copyright 1995-2003 CASEMaker Inc.

```
retcode = SQLBindCol(hstmt, 15, SQL_C_SHORT, &max_scale,  0, &len[15]);

/* tell odbc driver to get all type information                        */
printf("tell odbc driver to get all SQL type information \n");
SQLGetTypeInfo(hstmt,SQL_ALL_TYPES);

/ fetch all type information                                            */
do {
    retcode = SQLFetch(hstmt);
    switch (retcode)
           {
            case SQL_SUCCESS_WITH_INFO:
            case SQL_SUCCESS:
                 print type info(); /* print type info such as name,type,*/
                                    /* prec, prefix, ...             */
                 break;
            case SQL_NO_DATA_FOUND:
                 break;
            default:
                 print error
           }
   }while (retcode != SQL_NO_DATA_FOUND);
```

©Copyright 1995-2003 CASEMaker Inc.

SQLGetInfo

You can use SQLGetInfo to get general information about the data source.

Prototype

SQLGetInfo:

```
RETCODE  SQLGetInfo (
                     HDBC       hdbc,
                     UWORD      fInfoType,
                     PTR        rgbInfoValue,
                     SWORD      cbInfoValueMax,
                     SWORD FAR *pcbInfoValue)
```

Given a value in fInfoType representing the type of information you want to know, and given the output storage rgbInfoValue and its storage size cbInfoValueMax, SQLGetInfo will return the fetched information in rgbInfoValue and will return the size of the fetched information in pcbInfoValue.

Example 1

Checks if the data source supports the string function CONCAT:

```
UDWORD  bitmask;
SDWORD  retcode;
retcode = SQLGetInfo(hdbc, SQL STRING FUNCTIONS, (PTR) &bitmask,
                     sizeof(bitmask), NULL);
if (bitmask & SQL_FN_STR_CONCAT)
    printf ("the data source supports CONCAT\n");
else
    printf ("the data source does not support CONCAT\n");
```

If you want to know the maximum number of columns allowed in a table, you can try the code in example 2.

©Copyright 1995-2003 CASEMaker Inc.

➲ **Example 2**

Checks for the maximum number of columns permitted in a table:

```
UWORD  maxNCol;
SDWORD retcode;
retcode = SQLGetInfo(hdbc, SQL MAX COLUMNS IN TABLE, (PTR) &maxNCol,
                     sizeof(maxNCol));
printf ("In this data source, a table can have %d columns at most\n",
        (int) maxNCol );
```

SQLGetFunctions

You can use SQLGetFunctions to check what ODBC functions the data source supports.

➲ **Prototype**

SQLGetFunctions:

```
RETCODE  SQLGetFunctions (
                          HDBC hdbc,
                          UWORD fFunction,
                          UWORD FAR *pfExists)
```

The input argument fFunction specifies the kind of ODBC function. The value of fFunction can be **SQL_API_SQLCANCEL**, **SQL_API_SQLFETCH**, **SQL_PUTDATA**, etc. — **SQLCancel**, **SQLFetch**, **SQLPutData** are all ODBC functions.

For example, you can give the argument **SQL_API_SQLCANCEL** in fFunction to check whether the data source supports SQLCancel.

After submitting SQLGetFunctions, in order to determine the existence of the ODBC function, you can check the Boolean value(s) in pfExists, which is a pointer to a single Boolean value or a list of Boolean values.

©Copyright 1995-2003 CASEMaker Inc.

Example 1

Checks if the data source supports SQLExecDirect:

```
UWORD  fExecDirect;
SDWORD retcode;
retcode = SQLGetFunctions (hdbc, SQL_API_SQLEXECDIRECT, &fExecDirect);
if (fExecDirect)
   printf ("the data source supports SQLExecDirect\n");
else
   printf ("the data source does not support SQLExecDirect\n");
```

Example 2

Checks if the data source supports SQLTables:

```
UWORD  fExecDirect;
SDWORD retcode;
retcode = SQLGetFunctions (hdbc, SQL_API_SQLTABLES, &fExecDirect);
if (fExecDirect)
   printf ("the data source supports SQLExecDirect\n");
else
   printf ("the data source does not support SQLExecDirect\n");
```


©Copyright 1995-2003 CASEMaker Inc.

6.4 Procedure Information

You can use SQLProcedureColumns and SQLProcedures to get stored procedure information. These ODBC functions are illustrated by examples in the following sections

SQLProcedureColumns

You can use SQLProcedureColumns to retrieve information about the list of input and output parameters, as well as the content of the defined columns that make up the result set for the specified procedures. The driver returns the information as a result set.

➲ **Prototype**

SQLProcedureColumns:

```
RETCODE  SQLProcedureColumns (
                              HSTMT hstmt,
                              UCHAR *szProcQualifier,
                              SWORD   cbProcQualifier,
                              UCHAR  *szProcOwner,
                              SWORD   cbProcOwner,
                              UCHAR  *szProcName,
                              SWORD   cbProcName,
                              UCHAR  *szColumnName,
                              SWORD   cbColumnName)
```

A string to be used as a search pattern can exist in ***szProcQualifier***, ***szProcOwner***, and ***szProcName***. These three arguments and their corresponding string length arguments, ***cbProcQualifier***, ***cbProcOwner***, and ***cbProcName*** appear in SQL*Procedures*.

The arguments for SQLProcedureColumns are:

- *hstmt* — a valid statement handle for retrieved results.

©Copyright 1995-2003 CASEMaker Inc.

- *szProcQualifier* —not supported by DBMaker, and should be **NULL** or an empty string.
- *cbProcQualifier* — the length of szProcQualifier, and should be zero.
- *szProcOwner* — points to the owner name. The owner is the person who created the procedure. It can be a string to be used as a search pattern or a **NULL** value. Use a **NULL** value to indicate all owners.
- *cbProcOwner* — the length of szProcOwner or **SQL_NTS**.
- *szProcName* — points to the names of procedures. It can be a string to be used as a search pattern or a **NULL** value. Use a **NULL** value to indicate all procedures.
- *cbProcName* — the length of szProcName or **SQL_NTS**.
- *szColumnName* — points to the names of columns. It can be a string to be used as a search pattern or a NULL value. Use a NULL value to indicate all columns.
- *cbColumnName* — the length of szColumnName.

SQLProcedureColumns returns a result set consisting of the following columns:

Column No.	Column Name	Data Type	Comments
1	PROCEDURE_QUALIFIER	VARCHAR(128)	
2	PROCEDURE_OWNER	VARCHAR(128)	
3	PROCEDURE_NAME	VARCHAR(128)	NOT NULL
4	COLUMN_NAME	VARCHAR(128)	NOT NULL
5	COLUMN_TYPE	SMALLINT	NOT NULL
6	DATA_TYPE	SMALLINT	NOT NULL
7	TYPE_NAME	VARCHAR(128)	NOT NULL
8	PRECISION	INTEGER	
9	LENGTH	INTEGER	
10	SCALE	SMALLINT	
11	RADIX	SMALLINT	
12	NULLABLE	SMALLINT	NOT NULL
13	REMARK	VARCHAR(254)	

©Copyright 1995-2003 CASEMaker Inc.

The following uses SQLProcedureColumns to fetch all information about database user Tom's stored procedure "employee" that makes up the result set for the specific procedure.

➲ **Example**

```
UCHAR  catalog[30], schema[30], procName[30], colName[30];

UCHAR  typeName[30], remark[30];

SWORD  colType, dataType, scale, radix, nullable;

UDWORD prec, length, len[13];

SDWORD retcode;

/* bind all columns                                                    */

retcode = SQLBindCol(hstmt,  1, SQL_C_CHAR,  catalog,   30, &len[1]);

retcode = SQLBindCol(hstmt,  2, SQL_C_CHAR,  schema,    30, &len[2]);

retcode = SQLBindCol(hstmt,  3, SQL C CHAR,  procName,  30, &len[3]);

retcode = SQLBindCol(hstmt,  4, SQL_C_CHAR,  colName,   30, &len[4]);

retcode = SQLBindCol(hstmt,  5, SQL_C_SHORT, &colType,   0, &len[5]);

retcode = SQLBindCol(hstmt,  6, SQL C SHORT, &dataType,  0, &len[6]);

retcode = SQLBindCol(hstmt,  7, SQL C CHAR,  typeName,  30, &len[7]);

retcode = SQLBindCol(hstmt,  8, SQL_C_LONG,  &prec,      0, &len[8]);

retcode = SQLBindCol(hstmt,  9, SQL_C_LONG,  &length,    0, &len[9]);

retcode = SQLBindCol(hstmt, 10, SQL C SHORT,  &scale,    0, &len[10]);

retcode = SQLBindCol(hstmt, 11, SQL C SHORT,  &radix,    0, &len[11]);

retcode = SQLBindCol(hstmt, 12, SQL_C_SHORT, &nullable,  0, &len[12]);
retcode = SQLBindCol(hstmt, 13, SQL_C_CHAR,  remark     30, &len[13]);

retcode = SQLProcedureColumns(hstmt, null, 0, "Tom", SQL NTS,

                              "employee", SQL NTS, null, 0);

while ((retcode = SQLFetch(hstmt)) == SQL SUCCESS)

      { /* print out each column's content in the result set */

       printf("column 1 :  procedure qualifier = %s\n", catalog);

       printf("column 2 :  procedure owner = %s\n", schema);

       printf("column 3 :  procedure name = %s\n", procName);

       printf("column 4 :  column name = %s\n", colName);
```

©Copyright 1995-2003 CASEMaker Inc.

```
        printf("column 5 :  column type = %d\n", colType);
        printf("column 6 :  data type = %d\n", dataType);
        printf("column 7 :  type name = %s\n", typeName);
        printf("column 8 :  precision = %d\n", prec);
        printf("column 9 :  length = %d\n", length);
        printf("column 10 : scale = %d\n", scale);
        printf("column 11 : radix = %d\n", radix);
        printf("column 12 : nullable = %d\n", nullable);
        printf("column 13 : remark = %s\n", remark);
       }
...
```

SQLProcedures

You can use SQLProcedures to get the list of procedure names that is stored in the data source.

➲ **Prototype**

SQLProcedures:

```
RETCODE SQLProcedures (
                        HSTMT   hstmt,
                        UCHAR   *szProcQualifier,
                        SWORD   cbProcQualifier,
                        UCHAR   *szProcOwner,
                        SWORD   cbProcOwner,
                        UCHAR   *szProcName,
                        SWORD   cbProcName);
```

hstmt is a valid statement handle. szTableQualifier, cbTableQualifier, szTableOwner, cbTableOwner, szTableName, cbTableName are all defined the same as for **SQLProcedureColumns**. SQLProcedures returns the result as a standard result set, in the order PROCEDURE_QUALIFIER, PROCEDURE_OWNER, and PROCEDURE_NAME.

©Copyright 1995-2003 CASEMaker Inc.

The following table lists the columns in the result set:

Column No.	Column Name	Data Type	Comments
1	PROCEDURE_QUALIFIER	VARCHAR(128)	
2	PROCEDURE_OWNER	VARCHAR(128)	
3	PROCEDURE_NAME	VARCHAR(128)	NOT NULL
4	NUM_INPUT_PARAMS	N/A	
5	NUM_OUTPUT_PARAMS	N/A	
6	NUM_RESULT_SETS	N/A	
7	REMARKS	VARCHAR(254)	
8	PROCEDURE_TYPE	SMALLINT	

This following shows how to use SQLProcedures to fetch all procedures that created by “Tom”. If you want to retrieve all procedures in the database, you can use a null value in the field of the procedure owner.

➲ **Example**

```
UCHAR  catalog[30], schema[30], procName[30];
UCHAR  remark[30];
SWORD  type;
SDWORD len[5];
SDWORD retcode;
/* bind all columns                                                    */
retcode = SQLBindCol(hstmt,  1, SQL_C_CHAR,  catalog,  30, &len[1]);
retcode = SQLBindCol(hstmt,  2, SQL_C_CHAR,  schema,   30, &len[2]);
retcode = SQLBindCol(hstmt,  3, SQL C CHAR,  procName, 30, &len[3]);
retcode = SQLBindCol(hstmt,  7, SQL_C_CHAR,  remark,   30, &len[4]);
retcode = SQLBindCol(hstmt,  8, SQL_C_SHORT, &type,     0, &len[5]);
retcode = SQLProcedures(hstmt, null, 0,
                                “Tom”, SQL NTS,
                           null, 0);
while ((retcode = SQLFetch(hstmt)) == SQL SUCCESS)
      { /* print out each column’s content in the result set */
       printf("column 1 :  procedure qualifier = %s\n", catalog);
```

```
    printf("column 2 :  procedure owner = %s\n", schema);
    printf("column 3 :  procedure name = %s\n", procName);
    printf("column 7 :  remark = %s\n", remark);
    printf("column 8 :  type = %d\n", type);
   }
```


©Copyright 1995-2003 CASEMaker Inc.

7 Transaction Control

In this chapter, we will describe the concepts of transactions and savepoints and their characteristics. We will also show you how to use ODBC functions to end a transaction and setup options for transaction control.

In this chapter you will learn how to:

- Set and use two different commit modes, auto-commit and manual-commit, by using the *SQLSetConnectOption* and *SQLGetConnectOption* functions,
- Terminate a transaction by using the *SQLTransact* function. The effects that occur when a transaction is terminated are also explained.

©Copyright 1995-2003 CASEMaker Inc.

7.1 Transactions and Savepoints

A transaction is a sequence of one or more SQL statements that form a logical unit of work. Each SQL statement in the transaction performs part of a task and all of them are necessary for the task. Only when all SQL statements in the transaction are executed successfully we can treat the task as completed.

➲ **To manage a deposit in a bank account, a program should:**

1. Query the account table to make sure the account name is valid.
2. Query the branch table to make sure the branch number is valid.
3. Query the teller table to check if the teller exists.
4. Insert a record into the history table for this deposit.
5. Update the balance of this account name in the account table, and add the money for this deposit.
6. Update the balance of this teller in the teller table.
7. Update the balance of this branch in the branch table.

NOTE *These seven operations compose a complete transaction, and each operation is an* SQL *statement. If any one of these statements fails, the execution of this entire transaction must be discarded or inconsistencies in the data may result.*

➲ **The general flow of a transaction:**

1. Start a transaction.
2. Execute the statements.
3. Roll back the changes if any of the statements fail.
4. Commit the changes if all of the statements succeed.

When you connect to DBMaker, a transaction starts automatically. You can execute as many SQL statements as you need. After these SQL statements are processed, to commit the transaction including all changes made by DML operations (INSERT, DELETE or UPDATE), call the ODBC function SQLTransact with the option **SQL_COMMIT**. On the other hand, if you want to abort the transaction, you can

©Copyright 1995-2003 CASEMaker Inc.

call the ODBC function SQLTransact with the option **SQL_ROLLBACK**. After one transaction is terminated, DBMaker will automatically start a new transaction.

Sometimes if the transaction is very long, you can use savepoints to divide the long transaction into several parts so that it is easier to manage the whole transaction. A savepoint is a logical marker that can be declared at a specified point within the context of a transaction. Using a savepoint allows you to undo all updates after the specified point in a transaction without undoing the whole transaction.

For example, if you execute a transaction that is composed of fifteen SQL statements, and you mark a savepoint between the 10th and 11th statement, you can rollback to the savepoint if an error occurs while executing the 12th statement. Then you only have to correct the statement the error occurred in and redo the 10th, 11th, and 12th statement instead of redoing all the statements in current transaction.

➲ **Example**

```
statement 1;
...
statement 5;
SAVEPOINT SVP1;      -> point A: define the first savepoint
statement 6;
...
statement 10;
SAVEPOINT SVP2;      -> point B: define the second savepoint
statement 11;
statement 12;        -> error occurs
ROLLBACK TO SVP2;    -> point C: when error occurs, rollback to nearest
savepoint
/* at this point, all the statements before SVP2 are preserved */
/* only statement 11 and 12 need to be re-executed.            */
statement 13;
statement 14;
statement 15;
COMMIT WORK;         -> if all statements are ok, commit the transaction
```

©Copyright 1995-2003 CASEMaker Inc.

In this example, we can see how savepoints can help us manage a long transaction. In DBMaker, you can define up to 32 savepoints in a transaction. After the transaction is terminated, all the defined savepoints in this transaction will be cleared.

Note that the savepoint ID must be unique in a transaction, e.g. if you have defined a savepoint named ***SVP1*** at point A, you cannot define another savepoint named ***SVP1*** at point B. Another important fact to remember when using savepoints is that when you roll back to a pre-defined savepoint, all the savepoints defined after that point are discarded.

E.g. At point C in the above example, if you rollback to savepoint ***SVP1***, then ***SVP2*** is discarded and cannot be used any more. However, you can then define a new savepoint called ***SVP2***.

©Copyright 1995-2003 CASEMaker Inc.

7.2 Terminating a Transaction

As described in the previous section, you can use SQLTransact to commit or rollback a transaction.

➲ **Prototype**

SQLTransact:

```
RETCODE SQLTransact(
                    HENV        henv,
                    HDBC        hdbc,
                    UWORD       fType);
```

Where fType can be either **SQL_COMMIT** or **SQL_ROLLBACK**. As their names imply, **SQL_COMMIT** commits the transaction and **SQL_ROLLBACK** rolls back the transaction.

In DBMaker, unless the value of connection option **SQL_CB_MODE** is set to **SQL_CB_PRESERVE** after a transaction is terminated (either committed or rolled back), or the user rolls back to a defined savepoint, all pending results associated with the statement handles in the current connection handle are cleared.

➲ **Example**

```
SQLAllocEnv(&henv);
SQLAllocConnect(henv,&hdbc);
/* connect to a database */
SQLConnect(hdbc, ...)
SQLAllocStmt(hdbc, &hstmt1);
SQLAllocStmt(hdbc, &hstmt2);
..
/* fetch one tuple from account table */
SQLExecDirect(hstmt1, "select * from account", SQL NTS);
SQLBindCol(hstmt1, 1, ....)
```

©Copyright 1995-2003 CASEMaker Inc.

```
SQLBindCol(hstmt1, 2, ....)
SQLFetch(hstmt1);
/* fetch one tuple from branch table */
SQLExecDirect(hstmt2, "select * from branch", SQL_NTS);
SQLBindCol(hstmt2, 1, ....)
SQLBindCol(hstmt2, 2, ....)
SQLFetch(hstmt2);
/* Commit the transaction */
SQLTransact(henv, hdbc, SQL_COMMIT);
```

When a transaction is committed, the unfetched data in the result sets associated with hstmt1 and hstmt2 are cleared (suppose the account and branch tables have more than one row).

©Copyright 1995-2003 CASEMaker Inc.

7.3 Auto-Commit & Manual-Commit

In general, application programs want to control the termination of transactions. In this situation, manual-commit mode is needed.

ODBC defines many connection options, and one of them is **SQL_AUTOCOMMIT**. This connection option indicates whether auto-commit mode should be turned on or off. The default value of the **SQL_AUTOCOMMIT** option is on. This means that every statement is committed automatically.

➲ **Example 1**

To start transaction processing, turn off the **SQL_AUTOCOMMIT** option by using the SQLSetConnectOption ODBC function.

```
SQLSetConnectOption(hdbc, SQL_AUTOCOMMIT, SQL_AUTOCOMMIT_OFF)
```

The user controls the commit action after this function call. If the auto-commit mode is off and the transaction is not committed when the user calls SQLDisconnect, DBMaker will rollback the transaction and return a warning.

➲ **Example 2**

To get the value of the current auto-commit mode, use the SQLGetConnectOption :

```
SQLGetConnectOption(hdbc, SQL_AUTOCOMMIT, &optVal);
```

If the option value in optVal is **SQL_AUTOCOMMIT_ON**, then each SQL statement will be committed automatically after it is successfully executed.

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.

8 ODBC 3.0 Functions

The Application Program Interface (API) found in DBMaker is now ODBC 3.0 compatible. To be ODBC 3.0 compatible, some new functions have been added, some old functions have been deprecated, and other functions have been modified. As a result, some of the existing functions may behave differently than in previous versions of DBMaker, while the use of other functions is discouraged. For a full description of ODBC 3.0 functions and how to use them, refer to the “Microsoft ODBC 3.0 Programmer’s Reference”.

©Copyright 1995-2003 CASEMaker Inc.

8.1 Deprecated functions

The following functions or function argument values have been deprecated in the DBMaker API (ODBC 3.0). DBMaker currently retains these functions for purposes of backward compatibility, but does not guarantee that they will appear in future versions.

- SQLAllocConnect—SQLAllocConnect has been replaced by SQLAllocHandle function with HandleType SQL_HANDLD_DBC. You should use SQLAllocHandle for this function in the future.
- SQLAllocEnv—SQLAllocEnv has been replaced by SQLAllocHandle function with HandleType SQL_HANDLD_ENV. You should use SQLAllocHandle for this function in the future.
- SQLAllocStmt—SQLAllocStmt has been replaced by SQLAllocHandle function with HandleType SQL_HANDLD_STMT. You should use SQLAllocHandle for this function in the future.
- SQLColAttributes—SQLColAttribute has replaced the SQLColAttribute function. You should use SQLColAttribute instead of SQLColAttributes for this function in the future.
- SQLExtendedFetch—SQLFetchScroll has replaced the SQLExtendedFetch function. You should use SQLFetchScroll instead of SQLExtendedFetch for this function in the future.
- SQLFreeConnect—SQLFreeConnect has been replaced by SQLFreeHandle with HandleType SQL_HANDLE_DBC. You should use SQLFreeHandle for this function in the future.
- SQLFreeEnv—SQLFreeEnv has been replaced by SQLFreeHandle with HandleType SQL_HANDLE_ENV. You should use SQLFreeHandle for this function in the future.
- SQLFreeStmt—the SQL_DROP value of the Option argument in SQLFreeStmt has been replaced by SQLFreeHandle with HandleType

©Copyright 1995-2003 CASEMaker Inc.

SQL_HANDLE_STMT. You should use SQLFreeHandle for this function in the future.

- SQLGetConnectOption-- SQLGetConnectAttr has replaced the SQLGetConnectOption function. You should use SQLGetConnectAttr instead of SQLGetConnectOption for this function in the future.
- SQLGetStmtOption-- SQLGetStmtAttr has replaced the SQLGetStmtOption function. You should use SQLGetStmtAttr instead of SQLGetStmtOption for this function in the future.
- SQLSetConnectOption-- SQLSetConnectAttr has replaced the SQLSetConnectOption function. You should use SQLSetConnectAttr instead of SQLSetConnectOption for this function in the future.
- SQLSetPos—the SQL_ADD value of the fOption argument in the SQLSetPos function has been replaced by the SQL_ADD value of the Operation value in the SQLBulkOperations function. You should use SQLBulkOperations instead of SQLSetPos for this function in the future.
- SQLSetStmtOption-- SQLSetStmtAttr has replaced the SQLSetStmtOption function. You should use SQLSetStmtAttr instead of SQLSetStmtOption for this function in the future.
- SQLTransact—SQLTransact has been replaced by the SQLEndTran function. You should use SQLTransact instead of SQLEndTran for this function in the future.

©Copyright 1995-2003 CASEMaker Inc.

8.2 Modified functions

The following functions have been modified in the DBMaker API (ODBC 3.0). The behavior of these functions will differ somewhat from DBMaker 3.01 and earlier APIs (ODBC 2.0). However, the behavior of these functions will remain unchanged if you are using client software from versions of DBMaker before version 3.5.

SQLCancel

The SQLCancel function is fully supported in the DBMaker API (ODBC 3.0). In previous versions of DBMaker (ODBC 2.0), when there was no processing being done on a statement calling the SQLCancel function, it had the same effect as calling SQLFreeStmt with the SQL_CLOSE option. In DBMaker, no processing being done on a statement calling the SQLCancel function has no effect. If you have a cursor open and want to close it, you should call SQLCloseCursor instead of SQLCancel.

SQLColumns

The SQLColumns function is fully supported in the DBMaker API (ODBC 3.0). The SQLColumns function will now return 18 columns, regardless of whether the client is using the ODBC2.0 or 3.0 API. The following table lists the column names returned by this function for SQLColumns function for both DBMaker and for previous versions of DBMaker.

DBMaker 3.5 - 4.x (ODBC 3.0)	DBMaker 2.0x, 3.0x (ODBC 2.0)
TABLE_CAT	TABLE_QUALIFIER
TABLE_SCHEM	TABLE_OWNER
TABLE_NAME	TABLE_NAME
COLUMN_NAME	COLUMN_NAME
DATA_TYPE	DATA_TYPE
TYPE_NAME	TYPE_NAME
COLUMN_SIZE	PRECISION
BUFFER_LENGTH	LENGTH
DECIMAL_DIGITS	SCALE

©Copyright 1995-2003 CASEMaker Inc.

DBMaker 3.5 - 4.x (ODBC 3.0)	DBMaker 2.0x, 3.0x (ODBC 2.0)
NUM_PREC_RADIX	RADIX
NULLABLE	NULLABLE
REMARKS	—
COLUMN_DEF	—
SQL_DATA_TYPE	—
SQL_DATETIME_SUB	—
CHAR_OCTET_LENGTH	—
ORDINAL_POSITION	—
IS_NULLABLE	—

SQLFetch

The SQLFetch function is fully supported in the DBMaker API (ODBC 3.0). In DBMaker, the SQLFetch function can now support rowsets with multiple rows. Earlier versions of DBMaker only supported single-row operations with the SQLFetch function.

SQLGetData

The SQLGetData function is fully supported in the DBMaker API (ODBC 3.0). In DBMaker, the SQLGetData function can now support rowsets with multiple rows. Earlier versions of DBMaker only supported single-row operations with the SQLGetData function.

SQLGetFunctions

The SQLGetFunctions function is fully supported in the DBMaker API (ODBC 3.0). In DBMaker, you can call the SQLGetFunctions function with a value of SQL_API_ODBC3_ALL_FUNCTIONS or SQL_API_ALL_FUNCTIONS for the *FunctionId* parameter. SQL_API_ODBC3_ALL_FUNCTIONS is used by ODBC 3.0 applications to determine support for ODBC 3.0 or earlier functions, while SQL_API_ALL_FUNCTIONS is used by ODBC 2.0 applications to determine support for ODBC 2.0 or earlier functions.

©Copyright 1995-2003 CASEMaker Inc.

If the value of *FunctionId* is SQL_API_ODBC3_ALL_FUNCTIONS, *SupportedPtr* points to a 4000-bit bitmap that can be used to determine whether an ODBC 3.0 or earlier function is supported. You can use SQL_API_ODBC3_ALL_FUNCTIONS with either ODBC 3.0 or 2.0 drivers. If the value of *FunctionID* is SQL_API_ALL_FUNCTIONS, then *SupportedPtr* returns an array of 100 elements that you can use to determine whether an ODBC 2.0 function is supported.

SQLGetInfo

The SQLGetInfo function is now fully supported in the DBMaker API (ODBC 3.0).

SQLProcedureColumns

The SQLProcedureColumns function is now fully supported in the DBMaker API (ODBC 3.0). DBMaker SQLProcedureColumns function will now return 19 columns, regardless of whether the client is using the ODBC 2.0 or 3.0 API. The following table lists the column names returned by this function for SQLProcedureColumns function for both DBMaker and for previous versions of DBMaker.

DBMaker 3.5 - 4.x (ODBC 3.0)	DBMaker 2.0x, 3.0x (ODBC 2.0)
PROCEDURE_CAT	PROCEDURE _QUALIFIER
PROCEDURE _SCHEM	PROCEDURE _OWNER
PROCEDURE _NAME	PROCEDURE _NAME
COLUMN_NAME	COLUMN_NAME
COLUMN_TYPE	COLUMN_TYPE
DATA_NAME	DATA_NAME
TYPE_NAME	TYPE_NAME
COLUMN_SIZE	PRECISION
BUFFER_LENGTH	LENGTH
DECIMAL_DIGITS	SCALE
NUM_PREC_RADIX	RADIX
NULLABLE	NULLABLE
REMARKS	REMARK
COLUMN_DEF	—

©Copyright 1995-2003 CASEMaker Inc.

DBMaker 3.5 - 4.x (ODBC 3.0)	DBMaker 2.0x, 3.0x (ODBC 2.0)
SQL_DATA_TYPE	—
SQL_DATETIME_SUB	—
CHAR_OCTET_LENGTH	—
ORDINAL_POSITION	—
IS_NULLABLE	—

©Copyright 1995-2003 CASEMaker Inc.

8.3 New functions

This section lists all new functions in DBMaker, the supported options of each function, and whether the function fully or partially supports the ODBC 3.0 standard.

SQLAllocHandle

The SQLAllocHandle function is fully supported in the DBMaker API (ODBC 3.0). It is a generic function for allocating environment, connection, statement, or descriptor handles, and replaces the ODBC 2.0 functions SQLAllocConnect, SQLAllocEnv, and SQLAllocStmt.

Prototype

SQLAllocHandle:

```
RETCODE SQLAllocHandle(
                    SQLSMALLINT HandleType,
                    SQLHANDLE   InputHandle,
                    SQLHANDLE * OutputHandlePtr);
```

The following example uses the SQLAllocHandle function to allocate the environment, connection, and statement handles.

Example

```
SQLHANDLE  henv, hdbc, hstmt;
SQLRETURE  retcode;
retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
(void*)SQL OV ODBC3, 0);
retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
retcode = SQLConnect(hdbc, (SQLCHAR*)  "test", SQL_NTS,
                           (SQLCHAR*) "Sysadm", SQL_NTS,
                           (SQLCHAR*) "coffee", SQL_NTS);
```


©Copyright 1995-2003 CASEMaker Inc.

```
retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
…
```

SQLBulkOperations

The SQLBulkOperations function is fully supported in the DBMaker API (ODBC 3.0). The SQLBulkOperations function performs bulk insertions and bookmark operations, including update, delete, and fetch by bookmark operations.

➲ **Prototype**

SQLBulkOperations:

```
RETCODE SQLBulkOperations(
                    SQLHSTMT     StatementHandle,
                    SQLUSMALLINT Operation);
```

The following table shows a list of options for the SQLBulkOperations function, and whether they are supported.

Operation	Supported?
SQL_ADD	Y
SQL_UPDATE_BY_BOOKMARK	Y
SQL_DELETE_BY_BOOKMARK	Y
SQL_FETCH_BY_BOOKMARK	Y

The following uses SQLBulkOperations function with option value SQL_ADD to insert two rows of data into table Employee.

➲ **Example**

```
SQLRETCODE  retcode;
SQLHANDLE   hstmt;
SQLUINTEGER CustID[2];
SQLCHAR     Name[2][18], Address[2][100], Phone[2][11];
SQLINTEGER  CustIDInd[2], NameInd[2], AddressInd[2], PhoneIbd[2];
/* set necessary statement attributes */
retCode = SQLSetStmtAttr(hstmt, SQL_ATTR_CURSOR_TYPE, SQL_CURSOR_DYNAMIC);
```

©Copyright 1995-2003 CASEMaker Inc.

```
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_CONCURRENCY, SQL_CONCUR_LOCK);
retcode = SQLSetStmtAttr(hstmt, SQL ROW ARRAY SIZE, 2);
/* binding columns */
retcode = SQLBindCol(hstmt, 1, SQL_C_LONG, CustID,  0,  CustIDInd);
retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, Name,    18,  NameInd);
retcode = SQLBindCol(hstmt, 3, SQL C CHAR, Address, 100, AddressInd);
retcode = SQLBindCol(hstmt, 4, SQL_C_CHAR, Phone,   11,  PhoneInd);
/* execute a query  */
SQLExecDirect(hstmt, "select * from Customers", SQL_NTS);
/* prepare data for insertion */
CustID[0] = 1;
CustID[1] = 2;
strcpy(Name[0], "Jackson");
strcpy(Name[1], "Clinton");
strcpy(Address[0], "107 Castlewood, Cary, NC11256");
strcpy(Address[1], "305 N. Frances St., Madison, WI95868");
strcpy(Phone[0], "02-78923423");
strcpy(Phone[1], "03-7893933");
/* insert data */
retcode = SQLBulkOperations(hstmt, SQL_ADD);
…
```

SQLCloseCursor

The SQLCloseCursor function is fully supported in the DBMaker API (ODBC 3.0). The SQLCloseCursor function closes a cursor that is open on a statement, and discards pending results.

Prototype

SQLCloseCursor:

```
RETCODE SQLCloseCursor(
```


©Copyright 1995-2003 CASEMaker Inc.

```
                    SQLHSTMT     StatementHandle);
```

SQLColAttribute

The SQLColAttribute function is fully supported in the DBMaker API (ODBC 3.0). The SQLColAttribute function returns information about a column in a result set either as a character string, a 32-bit value, or an integer value.

➲ **Prototype**

SQLColAttribute:

```
RETCODE SQLColAttribute(
                    SQLHSTMT      StatementHandle,
                    SQLUSMALLINT  ColumnNumber,
                    SQLUSMALLINT  FieldIdentifier,
                    SQLPOINTER    CharacterAttributePtr,
                    SQLSMALLINT   BufferLength,
                    SQLSMALLINT * StringLengthPtr,
                    SQLPOINTER    NumericAttributePtr);
```

The following table lists the descriptor types returned by this function.

FIELD IDENTIFIER	PURPOSE
SQL_DESC_AUTO_UNIQUE_VALUE	Determines whether a column is a SERIAL column (SQL_TRUE) or not (SQL_FALSE).
SQL_DESC_BASE_COLUMN_NAME	The base column name.
SQL_DESC_BASE_TABLE_NAME	The base table name.
SQL_DESC_CASE_SENSITIVE	Determines whether collations and comparisons on a column are case-sensitive (SQL_TRUE) or not (SQL_FALSE).
SQL_DESC_CATALOG_NAME	The catalog of the table that contains the column.
SQL_DESC_CONCISE_TYPE	The concise data type for date, time, and interval data types.

©Copyright 1995-2003 CASEMaker Inc.

FIELD IDENTIFIER	PURPOSE
SQL_DESC_COUNT	The number of columns available in the result set.
SQL_DESC_DISPLAY_SIZE	The maximum number of characters required to display data from a column.
SQL_DESC_FIXED_PREC_SCALE	Determines whether the column has a fixed precision and non-zero scale (SQL_TRUE) or not (SQL_FALSE).
SQL_DESC_LABEL	The column label or title. If the column does not have a label, the column name is returned.
SQL_DESC_LENGTH	The maximum or actual character length of a character string or binary data type.
SQL_DESC_LITERAL_PREFIX	The characters that DBMaker recognizes as a prefix for a literal of the data type the column contains.
SQL_DESC_LITERAL_SUFFIX	The characters that DBMaker recognizes as a suffix for a literal of the data type the column contains.
SQL_DESC_LOCAL_TYPE_NAME	The localized (native language) name for a data type that may be different than the regular name.
SQL_DESC_NAME	The column alias. If the column does not have an alias, the column name is returned.
SQL_DESC_NULLABLE	Determines whether the column can contain NULL values (SQL_NULLABLE) or not (SQL_NO_NULLS). If it is not known whether the column can contain NULL values, the value SQL_NULLABLE_UNKNOWN is returned.

©Copyright 1995-2003 CASEMaker Inc.

FIELD IDENTIFIER	PURPOSE
SQL_DESC_NUM_PREX_RADIX	Contains; 2 if SQL_DESC_TYPE is an approximate numeric data type and SQL_DESC_PRECISION contains the number of bits, 10 if SQL_DESC_TYPE is an exact numeric data type and SQL_DESC_PRECISION contains the number of decimal digits, and 0 if SQL_DESC_TYPE contains a non-numeric data type.
SQL_DESC_OCTET_LENGTH	The length in bytes of a character string or binary data type.
SQL_DESC_PRECISION	The precision of a numeric data type.
SQL_DESC_SCALE	The scale of a numeric data type.
SQL_DESC_SCHEMA_NAME	The schema of the table that contains the column.
SQL_DESC_SEARCHABLE	Determines whether the column can be used with any comparison operator in a WHERE clause (SQL_PRED_SEARCHABLE), with all comparison operators except LIKE (SQL_PRED_BASIC), only with the LIKE predicate (SQL_PRED_CHAR), or if it cannot be used in a WHERE clause at all (SQL_PRED_NONE).
SQL_DESC_TABLE_NAME	The name of the table that contains the column.
SQL_DESC_TYPE	A numeric value that specifies the data type of the column.
SQL_DESC_TYPE_NAME	A string that specifies the name of the data type (data-source dependent).
SQL_DESC_UNNAMED	Determines whether the value in SQL_DESC_NAME is a column name/alias (SQL_DESC_NAMED) or not (SQL_DESC_UNNAMED).

©Copyright 1995-2003 CASEMaker Inc.

FIELD IDENTIFIER	PURPOSE
SQL_DESC_UNSIGNED	Determines whether the column is unsigned (SQL_TRUE) or not (SQL_FALSE).
SQL_DESC_UPDATABLE	Describes if the column in the result set can be updated or not (not the column in the base table).

SQLCopyDesc

The SQLCopyDesc function is fully supported in the DBMaker API (ODBC 3.0). The SQLCopyDesc function copies descriptor information from one descriptor handle to another.

Prototype

SQLCopyDesc:

```
RETCODE SQLCopyDesc(
                    SQLHDESC      SourceDescHandle,
                    SQLHDESC      TargetDescHandle);
```

In the following example, descriptor operations are used to copy fields of the PartInfo table into Backup table. To do so, copy the fields of the IRD of hstmt1 to the fields of the IPD in hstmt2, and the fields of the ARD of hstmt1 to the fields of the APD in hstmt2.

Example

```
/* the structure of a record row */
typedef struct{
        SQLINTEGER  PartID;
        SQLINTEGER  PartIDInd;
        SQLUCHAR    Description[100];
        SQLINTEGER  DescriptionInd;
        DOUBLE      Price;
        SQLINTEGER  PriceInd;
```


©Copyright 1995-2003 CASEMaker Inc.

```
}PartInfo;

PartInfo parts[20];

SQLHANDLE  hstmt1, hstmt2;

SQLHANDLE  ird1, ard1, ipd2, apd2;

SQLRETCODE retcode;

/* get ARD and IRD of hstmt1 */

SQLGetStmtAttr(hstmt1, SQL_ATTR_APP_ROW_DESC, &ard1, 0, NULL);

SQLGetStmtAttr(hstmt1, SQL_ATTR_IMP_ROW_DESC, &ird1, 0, NULL);

/* get APD and IPD of hstmt2 */

SQLGetStmtAttr(hstmt2, SQL ATTR APP PARAM DESC, &apd2, 0, NULL);

SQLGetStmtAttr(hstmt2, SQL_ ATTR_IMP_PARAM_DESC, &ipd2, 0, NULL);

/* set necessary statement attributes on hstmt1 */

SQLSetStmtAttr(hstmt1, SQL_ATTR_ROW_BIND_TYPE, (SQLPOINTER)sizeof(PartInfo),
0);

SQLSetStmtAttr(hstmt1, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)20, 0);

/* execute a select statement */

SQLExecDirect(hstmt1, "select * from PartInfo", SQL_NTS);

/* binding columns */

SQLBindCol(hstmt1, 1, SQL_C_LONG, &parts[0].PartID,     0,
&parts[0].PartIDInd);

SQLBindCol(hstmt1, 2, SQL C CHAR, parts[0].Description, 100,
&parts[0].DescriptionInd);

SQLBindCol(hstmt1, 3, SQL_C_DOUBLE, &parts[0].Price,    0,
&parts[0].PriceInd);

/* calling SQLCopyDesc */

SQLCopyDesc(ard1, arp2);

SQLCopyDesc(ird1, ipd2);

/* prepare an insert statement on hstmt2 */

SQLPrepare(hstmt2, "insert into Backup values(?,?,?)", SQL NTS);

retcode = SQLFetchScroll(hstmt1, SQL FETCH NEXT, 0);

while(retcode = SQL_SUCCESS){

        SQLExecute(hstmt2);
```

©Copyright 1995-2003 CASEMaker Inc.

```
        retcode = SQLFetchScroll(hstmt1, SQL_FETCH_NEXT, 0);
}
…
```

SQLEndTran

The SQLEndTran function is fully supported in the DBMaker API (ODBC 3.0). The SQLEndTran requests the DBMaker server to perform a commit or rollback for all active operations on all statements associated with a connection, or for all connections associated with an environment. DBMaker supports the following values for the *CompletionType* argument: SQL_COMMIT and SQL_ROLLBACK.

Prototype

SQLEndTran:

```
RETCODE SQLEndTran(
                    SQLSMALLINT HandleType,
                    SQLHANDLE   Handle,
                    SQLSMALLINT CompletionType);
```

SQLFetchScroll

The SQLFetchScroll function is fully supported in the DBMaker API (ODBC 3.0). The SQLFetchScroll function fetches a rowset of data, at a relative or absolute position, or by a bookmark, from the result set and returns all bound columns. DBMaker supports the following values for the *FetchOrientation* argument: SQL_FETCH_NEXT, SQL_FETCH_PRIOR, SQL_FETCH_FIRST, SQL_FETCH_LAST, SQL_FETCH_ABSOLUTE, SQL_FETCH_BOOKMARK, and SQL_FETCH_RELATIVE.

Prototype

SQLFetchScroll:

```
RETCODE SQLFetchScroll(
                    SQLHSTMT    StatementHandle,
```


©Copyright 1995-2003 CASEMaker Inc.

```
                    SQLSMALLINT Handle,
                    SQLINTEGER  FetchOffset);
```

The following code fragment shows how to use SQLFetchScroll function with option SQL_NEXT to fetch the whole result set. To get the row status, you need to use SQLSetStmtAttr function with SQL_ATTR_ROW_STATUS_PTR, which is different from SQLExtendedFetch.

➲ **Example**

```
#define LENGTH 18
#define ROWSET_SIZE 5
SQLHandle hstmt;
RETCODE   retcode;
SQLUCHAR  empid[ROWSET_SIZE][LENGTH]
SQLUCHAR  name[ROWSET_SIZE][LENGTH];
FLOAT    salary[ROWSET_SIZE];
SQLINTEGER   empidInd[ROWSET SIZE], nameInd[ROWSET SIZE],
salaryInd[ROWSET_SIZE];
SQLUSMALLINT status[ROWSET_SIZE];
SQLUSMALLINT i;
/* set row status pointer */
retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, status, 0);
/* set rowset size */
retcode = SQLSetStmtAttr(hstmt, SQL ROW ARRAY SIZE, ROWSET SIZE);
/* execute a statement */
SQLExecDirect(hstmt, "SELECT * FROM EMPLOYEE", SQL_NTS);
/* binding columns */
SQLBindCol(hstmt, 1, SQL C CHAR, empid, LENGTH, empidInd);
SQLBindCol(hstmt, 2, SQL C CHAR, name, LENGTH, nameInd);
SQLBindCol(hstmt, 3, SQL_C_FLOAT, salary, 0, salaryInd);
retcode = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
for (i = 0; i < ROWSET_SIZE; i++)
```

©Copyright 1995-2003 CASEMaker Inc.

```
  {    if (status[i] == SQL_ROW_SUCCESS)
      {
        printf("tuple %ld  is - Employee ID : %s, Employee Name : %s, Salary
: %f \n", i+1, empid[i], name[i], salary[i]);
      }
    else {
        printf("fetch tuple %ld error \n", i+1);
      }
  }
…
```

SQLForeignKeys

The SQLForeignKeys function is fully supported in the DBMaker API (ODBC 3.0). The SQLForeignKeys function returns a list of foreign keys in the specified table, or a list of foreign keys in other tables that reference the primary key in the specified table.

➲ **Prototype**

SQLForeignKeys:

```
RETCODE SQLForeignKeys(
                  SQLHSTMT    StatementHandle,
                  SQLCHAR *   PKCatalogName,
                  SQLSMALLINT NameLength1,
                  SQLCHAR *   PKSchemaName,
                  SQLSMALLINT NameLength2,
                  SQLCHAR *   PKTableName,
                  SQLSMALLINT NameLength3,
                  SQLCHAR *   FKCatalogName,
                  SQLSMALLINT NameLength4,
                  SQLCHAR *   FKSchemaName,
                  SQLSMALLINT NameLength5,
                  SQLCHAR *   FKTableName,
                  SQLSMALLINT  NameLength6);
```


©Copyright 1995-2003 CASEMaker Inc.

This example uses two tables: ORDER (ORDERID, CUSTID, OPENDATE) and CUSTOMER (CUSTID, NAME, ADDRESS, PHONE).

In the *ORDER* table, CUSTID identifies the customer to whom the sale has been made. A foreign key refers to ORDERID in the CUSTOMER table.

This example calls SQLForeignKeys to get foreign keys in other table that reference the primary key of the ORDER table.

➲ **Example**

```
#define TAB LEN 18
#define COL_LEN 18
SQLUCHAR     pkTable[TAB_LEN+1], fkTable[TAB_LEN+1];
SQLUCHAR     pkCol[COL LEN+1],   fkCol[COL LEN+1];
SQLHANDLE    hstmt;
SQLINTEGER   pkTableInd, fkTableInd, pkColInd, fkColInd;
SQLRETCODE   retcode;
/* Bind the columns that describe the primary and foreign keys */
SQLBindCol(hstmt, 3, SQL C CHAR, pkTable, TAB LEN, &pkTableInd);
SQLBindCol(hstmt, 4, SQL_C_CHAR, pkCol,   COL_LEN, &pkColInd);
SQLBindCol(hstmt, 7, SQL_C_CHAR, fkTable, TAB_LEN, &fkTableInd);
SQLBindCol(hstmt, 8, SQL C CHAR, fkCol,   COL LEN, &fkColInd);
/* Get the names of columns in the primary key. */
retcode = SQLForeignKeys(hstmt, NULL, 0, NULL, 0, "ORDER", SQL_NTS,
                         NULL, 0, NULL, 0, NULL, 0);
while (retcode == SQL SUCCESS || retcode == SQL SUCCESS WITH INFO){
        retcode = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
        printf("Primary Table : %s, Primary Column : %s \n", pkTable, pkCol);
        printf("Foreign Table : %s, Foreign Column : %s \n", fkTable, fkCol);
}
/* close the cursor */
SQLCloseCursor(hstmt);
```

©Copyright 1995-2003 CASEMaker Inc.

...

SQLFreeHandle

The SQLFreeHandle function is fully supported in the DBMaker API (ODBC 3.0). The SQLFreeHandle function frees resources associated with an environment, connection, statement, or descriptor handle that was previously allocated using the SQLAllocHandle function.

©Copyright 1995-2003 CASEMaker Inc.

➲ **Prototype**

SQLFreeHandle:

```
RETCODE SQLFreeHandle(
                    SQLSMALLINT HandleType,
                    SQLHANDLE   Handle);
```

The following code fragment shows how to use SQLFreeHandle function to free the environment handle.

➲ **Example**

```
SQLHANDLE   henv;
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
…
SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

SQLGetConnectAttr

The SQLGetConnectAttr function is partially supported in the DBMaker API (ODBC 3.0). The SQLSetConnectAttr function gets attributes for a database connection. This function replaces the SQLGetConnectOption function in ODBC 2.0.

➲ **Prototype**

SQLGetConnectAttr:

```
RETCODE SQLGetConnectAttr(
                           SQLHDBC      ConnectionHandle,
                           SQLINTEGER   Attribute,
                           SQLPOINTER   ValuePtr,
                           SQLINTEGER   BufferLength,
                           SQLINTEGER  *StringLengthPtr);
```

The following table lists the attributes that can be retrieved using this function, and whether they are supported in DBMaker.

©Copyright 1995-2003 CASEMaker Inc.

Attribute	Supported?
SQL_ATTR_ACCESS_MODE	Y
SQL_ATTR_ASYNC_ENABLE	Y
SQL_ATTR_AUTO_IPD	Y
SQL_ATTR_AUTOCOMMIT	Y
SQL_ATTR_CONNECTION_TIMEOUT	Y
SQL_ATTR_CURRENT_CATALOG	Y
SQL_ATTR_LOGIN_TIMEOUT	Y
SQL_ATTR_METADATA_ID	Y
SQL_ATTR_ODBC_CURSORS	N
SQL_ATTR_PACKET_SIZE	N
SQL_ATTR_QUIET_MODE	N
SQL_ATTR_TRACE	N
SQL_ATTR_TRACEFILE	N
SQL_ATTR_TRANSLATE_LIB	N
SQL_ATTR_TRANSLATE_OPTION	N
SQL_ATTR_TXN_ISOLATION	Y
SQL_ATTR_QUERY_TIMEOUT	Y
SQL_ATTR_MAX_ROWS	Y

SQLGetDescField

The SQLGetDescField function is fully supported in the DBMaker API (ODBC 3.0). The SQLGetDescField function gets the value of a single field of a descriptor record.

➲ **Prototype**

SQLGetDescField:

```
RETCODE SQLGetDescField(
                    SQLHDESC     DescriptorHandle,
                    SQLSMALLINT  RecNumber,
                    SQLSMALLINT  FieldIdentifier,
                    SQLPOINTER   ValuePtr,
                    SQLINTEGER   BufferLength,
                    SQLINTEGER * StringLengthPtr);
```


©Copyright 1995-2003 CASEMaker Inc.

The following table lists the descriptor fields that can be retrieved using this function in DBMaker. In the table, the "G" represents Get, *S* represents Set, and "I" represents Invalid. SQL_DESC_ARRAY_SIZE only supports a value of one.

FieldIdentifier	ARD	APD	IRD	IPD
SQL_DESC_ALLOC_TYPE	G	G	G	G
SQL_DESC_ARRAY_SIZE	G/S	G/S	I	I
SQL_DESC_ARRAY_STATUS_PTR	G/S	G/S	G/S	G/S
SQL_DESC_BIND_OFFSET_PTR	G/S	G/S	I	I
SQL_DESC_BIND_TYPE	G/S	G/S	I	I
SQL_DESC_COUNT	G/S	G/S	G	G/S
SQL_DESC_ROW_PROCESSED_PTR	I	I	G/S	G/S
SQL_DESC_AUTO_UNIQUE_VALUE	I	I	G	I
SQL_DESC_BASE_COLUMN_NAME	I	I	G	I
SQL_DESC_BASE_TABLE_NAME	I	I	G	I
SQL_DESC_CASE_SENSITIVE	I	I	G	I
SQL_DESC_CATALOG_NAME	I	I	G	I
SQL_DESC_CONCISE_TYPE	G/S	G/S	G	G/S
SQL_DESC_DATA_PTR	G/S	G/S	I	I
SQL_DESC_DATETIME_INTERVAL_ CODE	G/S	G/S	G	G/S
SQL_DESC_DATETIME_INTERVAL_ PRECISION	G/S	G/S	G	G/S
SQL_DESC_DISPLAY_SIZE	I	I	G	I
SQL_DESC_FIXED_PREC_SCALE	I	I	G	G
SQL_DESC_INDICATOR_PTR	G/S	G/S	I	I
SQL_DESC_LABLE	I	I	G	I
SQL_DESC_LENGTH	G/S	G/S	G	G/S
SQL_DESC_LITERAL_PREFIX	I	I	G	I
SQL_DESC_LITERAL_SU FFIX	I	I	G	I
SQL_DESC_LOCAL_TYPE_NAME	I	I	G	G
SQL_DESC_NAME	I	I	G	G
SQL_DESC_NULLABLE	I	I	G	G
SQL_DESC_NUM_PREC_RADIX	G/S	G/S	G	G/S
SQL_DESC_OCTET_LENGTH	G/S	G/S	G	G/S
SQL_DESC_OCTET_LENGTH_PTR	G/S	G/S	I	I
SQL_DESC_PARAMETER_TYPE	I	I	I	G/S
SQL_DESC_PRECISION	G/S	G/S	G	G/S

©Copyright 1995-2003 CASEMaker Inc.

FieldIdentifier	ARD	APD	IRD	IPD
SQL_DESC_SCALE	G/S	G/S	G	G/S
SQL_DESC_SCHEMA_NAME	I	I	G	I
SQL_DESC_SEARCHABLE	I	I	G	I
SQL_DESC_TABLE_NAME	I	I	G	I
SQL_DESC_TYPE	G/S	G/S	G	G/S
SQL_DESC_TYPE_NAME	I	I	G	G
SQL_DESC_UNNAMED	I	I	G	I
SQL_DESC_UNSIGNED	I	I	G	G
SQL_DESC_UPDATABLE	I	I	G	I

The following code fragment shows how to use SQLGetDescField to get the columns' information.

Example

```
#define LEN 19
SQLHANDLE   hstmt, ird;
SQLINTEGER  count, index;
SQLUCHAR    colName[LEN], typeName[LEN];
SQLSMALLINT prec, scale;
SQLUINTEGER length;
/* execute the statement */
SQLExecDirect(hstmt, "select * from EMPLOYEE", SQL NTS);
/* get the ird descriptors */
 SQLGetStmtAttr(hstmt, SQL_ATTR_IMP_ROW_DESC, &ird, SQL_IS_POINTER, NULL);
/* get the number of columns */
SQLGetDescField(ird, 0, SQL DESC COUNT, &count, 0, NULL);
for (index = 1; index <= count; index++)
{
        SQLGetDescField(ird, index, SQL DESC NAME, colName, LEN, NULL);
        SQLGetDescField(ird, index, SQL_DESC_TYPE_NAME, typeName, LEN, NULL);
        SQLGetDescField(ird, index, SQL_DESC_PRECISION, &prec, 0, NULL);
        SQLGetDescField(ird, index, SQL_DESC_SCALE, &scale, 0, NULL);
```


©Copyright 1995-2003 CASEMaker Inc.

```
        SQLGetDescField(ird, index, SQL_DESC_LENGTH, &len, 0, NULL);

        printf("Column No : %d, Name : %s, Type : %s, Length : %d, Precision
: %d, Scale : %d \n", index, colName, typeName, len, prec, scale);

}

…
```

SQLGetDescRec

The SQLGetDescRec function is fully supported in the DBMaker API (ODBC 3.0). The SQLGetDescRec function returns the settings or values of multiple fields in a descriptor record.

➲ **Prototype**

SQLGetDescRec:

```
RETCODE SQLGetDescRec(
                  SQLHDESC      DescriptorHandle,
                  SQLSMALLINT   RecNumber,
                  SQLCHAR*      Name,
                  SQLSMALLINT   BufferLength,
                  SQLSMALLINT*  StringLengthPtr,
                  SQLSMALLINT*  TypePtr,
                  SQLSMALLINT*  SubTypePtr,
                  SQLINTEGER*   LengthPtr,
                  SQLSMALLINT*  PrecisionPtr,
                  SQLSMALLINT*  ScalePtr,
                  SQLSMALLINT*  NullablePtr);
```

The following code fragment shows how to use SQLGetDescRec function to get column information.

➲ **Example**

```
#define LEN 19

SQLHANDLE   hstmt, ird;
```

©Copyright 1995-2003 CASEMaker Inc.

```
SQLINTEGER  count, index;

SQLUCHAR    colName[LEN];

SQLSMALLINT type, prec, scale, nullable;

SQLUINTEGER length;

/* execute the statement */

SQLExecDirect(hstmt, "select * from EMPLOYEE", SQL NTS);

/* get the ird descriptors */

 SQLGetStmtAttr(hstmt, SQL_ATTR_IMP_ROW_DESC, &ird, SQL_IS_POINTER, NULL);

/* get the number of columns */

SQLGetDescField(ird, 0, SQL DESC COUNT, &count, 0, NULL);

for (index = 1; index <= count; index++)

{

        SQLGetDescRec(ird, index, colName, LEN, NULL, &type, NULL, &len,
&prec, &scale, &nullable);

        printf("Column No. : %d, Name : %s, Type : %d, Length : %d, Precision
: %d, Scale : %d, Nullable : %d \n", index, colName, type, len, prec, scale,
null);

}

…
```

SQLGetDiagField

The SQLGetDiagField function is partially supported in the DBMaker API (ODBC 3.0). The SQLGetDiagField function returns the current value of a field from a record in a specified handle's diagnostic data structure. The field contains error, warning, and status information.

Prototype

SQLGetDiagField:

```
RETCODE SQLGetDiagField(

                   SQLSMALLINT   HandleType,

                   SQLHANDLE     Handle,

                   SQLSMALLINT   RecNumber,
```


©Copyright 1995-2003 CASEMaker Inc.

```
                    SQLSMALLINT   DiagIdentifier,
                    SQLPOINTER    DiagInfoPtr,
                    SQLSMALLINT   BufferLength,
                    SQLSMALLINT*  StringLengthPtr);
```

The following table shows the identifier of the field required from the diagnostic data structure, and whether it is supported in DBMaker.

DiagIdentifier	Supported?
SQL_DIAG_CURSOR_ROW_COUNT	N
SQL_DIAG_DYNAMIC_FUNCTION	N
SQL_DIAG_DYNAMIC_FUNCTION_CODE	N
SQL_DIAG_NUMBER	Y
SQL_DIAG_RETURNCODE	Y
SQL_DIAG_ROW_COUNT	Y
SQL_DIAG_CLASS_ORIGIN	Y
SQL_DIAG_CONNECTION_NAME	Y
SQL_DIAG_MESSAGE_TEXT	Y
SQL_DIAG_SERVER_NAME	Y
SQL_DIAG_SQLSTATE	Y
SQL_DIAG_SUBCLASS_ORIGIN	Y
SQL_DIAG_COLUMN_NUMBER	N
SQL_DIAG_NATIVE	N
SQL_DIAG_ROW_NUMBER	N

The following code fragment shows how to use SQLGetDiagField function and SQLGetDiagRec function to get the error information.

➲ **Example**

```
SQLHANDLE   hstmt;
SQLRETCODE  retcode;
SQLINTEGER  num, index, nativerc;
SQLUCHAR    state[20], errmsg[200];
SQLSMALLINT retLen;
/* execute a statement */
```

©Copyright 1995-2003 CASEMaker Inc.

```
retcode = SQLExecDirect(hstmt, "SELECT * FROM EMPLOYEE", SQL_NTS);

/* if error, get error info */

if (retcode != SQL SUCCESS){

        SQLGetDiagField(SQL_HANDLE_STMT, hstmt, 0, SQL_DIAG_NUM, &num, 0,
NULL);

        for (index = 1; index <= num; index++){

             SQLGetDiagRec(SQL_HANDLE_STMT, hstmt, index, state, &nativerc,
errmsg, 200, &retLen);

        }

}

….
```

SQLGetDiagRec

The SQLGetDiagRec function is fully supported in the DBMaker API (ODBC 3.0). The SQLGetDiagRec function returns the values of several commonly used fields of a diagnostic record, including SQLSTATE, the native error code, and the diagnostic message text. DBMaker supports the following values for the *HandleType* argument: SQL_HANDLE_ENV, SQL_HANDLE_DBC, SQL_HANDLE_STMT, and SQL_HANDLE_DESC.

➲ **Prototype**

SQLGetDiagRec:

```
RETCODE SQLGetDiagRec(
                   SQLSMALLINT   HandleType,
                   SQLHANDLE     Handle,
                   SQLSMALLINT   RecNumber,
                   SQLCHAR *     Sqlstate,
                   SQLINTEGER *  NativeErrorPtr,
                   SQLCHAR *     MessageText,
                   SQLSMALLINT   BufferLength,
                   SQLSMALLINT * TextLengthPtr);
```


©Copyright 1995-2003 CASEMaker Inc.

SQLGetEnvAttr

The SQLGetEnvAttr function is partially supported in the DBMaker API (ODBC 3.0). The SQLGetEnvAttr function gets attributes for an environment handle.

➲ **Prototype**

SQLGetEnvAttr:

```
RETCODE SQLSetEnvAttr(
                    SQLHENV      EnvironmentHandle,
                    SQLINTEGER   Attribute,
                    SQLPOINTER   ValuePtr,
                    SQLINTEGER   BufferLength,
                    SQLINTEGER * StringLengthPtr);
```

The following table lists the attributes that can be retrieved using this function, and whether they are supported in DBMaker.

Attribute	Supported?
SQL_ATTR_ODBC_VERSION	Y
SQL_ATTR_OUTPUT_NTS	Y
SQL_ATTR_CONNECTION_POOLING	N
SQL_ATTR_CP_MATCH	N

SQLGetStmtAttr

The SQLGetStmtAttr function is partially supported in the DBMaker API (ODBC 3.0). The SQLGetStmtAttr function sets attributes for a statement. This function replaces the SQLGetStatementOption function in ODBC 2.0.

➲ **Prototype**

SQLGetStmtAttr:

```
RETCODE SQLGetStmtAttr(
                    SQLHSTMT     StatementHandle,
                    SQLINTEGER   Attribute,
```

©Copyright 1995-2003 CASEMaker Inc.

```
                    SQLPOINTER   ValuePtr,
                    SQLINTEGER   BufferLength,
                    SQLINTEGER * StringLengthPtr);
```

The following table lists the attributes that can be retrieved using this function, and whether they are supported in DBMaker.

Attribute	Supported?
SQL_ATTR_APPPARAM_DESC	Y
SQL_ATTR_APP_ROW_DESC	Y
SQL_ATTR_ASYNC_ENABLE	Y
SQL_ATTR_CONCURRENCY	Y
SQL_ATTR_CURSOR_SCROLLABLE	Y
SQL_ATTR_CURSOR_SENSITIVITY	Y
SQL_ATTR_CURSOR_TYPE	Y
SQL_ATTR_ENABLE_AUTO_IPD	Y
SQL_ATTR_FETCH_BOOKMARK_PTR	Y
SQL_ATTR_IMP_PARAM_DESC	Y
SQL_ATTR_IMP_ROW_DESC	Y
SQL_ATTR_KEYSET_SIZE	Y
SQL_ATTR_MAX_LENGTH	Y
SQL_ATTR_MAX_ROWS	Y
SQL_ATTR_METADATA_ID	Y
SQL_ATTR_NOSCAN	Y
SQL_ATTR_PARAM_BIND_OFFSET_PTR	Y
SQL_ATTR_PARAM_BIND_TYPE	Y
SQL_ATTR_PARAM_OPERATION_PTR	Y
SQL_ATTR_PARAM_STATUS_PTR	Y
SQL_ATTR_PARAMS_PROCESSED_PTR	Y
SQL_ATTR_PARAMSET_SIZE	Y
SQL_ATTR_QUERY_TIMEOUT	Y
SQL_ATTR_RETRIEVE_DATA	Y
SQL_ATTR_ROW_ARRAY_SIZE	Y
SQL_ATTR_ROW_BIND_OFFSET_PTR	Y
SQL_ATTR_ROW_BIND_TYPE	Y
SQL_ATTR_ROW_NUMBER	N
SQL_ATTR_ROW_OPERATION_PTR	Y

©Copyright 1995-2003 CASEMaker Inc.

Attribute	Supported?
SQL_ATTR_ROW_STATUS_PTR	Y
SQL_ATTR_ROWS_FETCHED_PTR	Y
SQL_ATTR_SIMULATE_CURSOR	Y
SQL_ATTR_USE_BOOKMARK	Y

SQLPrimaryKeys

The SQLPrimaryKeys function is fully supported in the DBMaker API (ODBC 3.0). The SQLPrimaryKeys function returns the column names that make up the primary key of a table as a result set.

➲ **Prototype**

SQLPrimaryKeys:

```
RETCODE SQLPrimaryKeys(
                    SQLHSTMT      StatementHandle,
                    SQLCHAR *     CatalogName,
                    SQLSMALLINT   NameLength1,
                    SQLCHAR *     SchemaName,
                    SQLSMALLINT   NameLength2,
                    SQLCHAR *     TableName,
                    SQLSMALLINT   NameLength3);
```

This example uses the table CUSTOMER (CUSTID, NAME, ADDRESS, PHONE), and CUSTID is the primary key in the CUSTOMER table.

This example calls SQLPrimaryKeys to get primary keys information of the CUSTOMER table.

➲ **Example**

```
#define TAB_LEN 19
#define COL LEN 19
SQLUCHAR     pkTable[TAB LEN], fkTable[TAB LEN];
SQLHANDLE    hstmt;
```

```
SQLINTEGER   pkTableInd, pkColInd;
SQLRETCODE   retcode;
/* Bind the columns that describe the primary and foreign keys */
SQLBindCol(hstmt, 3, SQL_C_CHAR, pkTable, TAB_LEN, &pkTableInd);
SQLBindCol(hstmt, 4, SQL_C_CHAR, pkCol,   COL_LEN, &pkColInd);
/* Get the names of columns in the primary key. */
retcode = SQLPrimaryKeys(hstmt, NULL, 0, NULL, 0, "CUSTOMER", SQL_NTS);
while (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO){
        retcode = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
        printf("Table : %s, Column : %s \n", pkTable, pkCol);
}
/* close the cursor */
SQLCloseCursor(hstmt);
…
```

SQLSetConnectAttr

The SQLSetConnectAttr function is partially supported in the DBMaker API (ODBC 3.0). The SQLSetConnectAttr function sets attributes for a database connection. The function replaces the SQLSetConnectOption function in ODBC 2.0. **The function prototype for SQLSetConnectAttr is:**

```
RETCODE SQLSetConnectAttr(
                   SQLHDBC    ConnectionHandle,
                   SQLINTEGER Attribute,
                   SQLPOINTER ValuePtr,
                   SQLINTEGER StringLength);
```

The following table lists the attributes that can be set using this function, and whether they are supported in DBMaker.

The following attributes apply: SQL_ATTR_ASYNC_ENABLE only supports SQL_ASYNC_ENABLE_OFF. SQL_ATTR_CONNECTION_TIMEOUT only

©Copyright 1995-2003 CASEMaker Inc.

supports a value of *0* (no time-out). SQL_ATTR_METADATA_ID only supports *SQL_FALSE*. SQL_ATTR_MAX_ROWS only supports a value of *0* (all rows).

Attribute	Supported?
SQL_ATTR_ACCESS_MODE	Y
SQL_ATTR_ASYNC_ENABLE	Y
SQL_ATTR_AUTO_IPD	Y
SQL_ATTR_AUTOCOMMIT	Y
SQL_ATTR_CONNECTION_TIMEOUT	Y
SQL_ATTR_CURRENT_CATALOG	Y
SQL_ATTR_LOGIN_TIMEOUT	Y
SQL_ATTR_METADATA_ID	Y
SQL_ATTR_ODBC_CURSORS	N
SQL_ATTR_PACKET_SIZE	N
SQL_ATTR_QUIET_MODE	N
SQL_ATTR_TRACE	N
SQL_ATTR_TRACEFILE	N
SQL_ATTR_TRANSLATE_LIB	N
SQL_ATTR_TRANSLATE_OPTION	N
SQL_ATTR_TXN_ISOLATION	Y
SQL_ATTR_QUERY_TIMEOUT	Y
SQL_ATTR_MAX_ROWS	Y

SQLSetDescField

The SQLSetDescField function is fully supported in the DBMaker API (ODBC 3.0). The SQLSetDescField function sets the value of a single field of a descriptor record.

➲ **Prototype**

SQLSetDescField:

```
RETCODE SQLSetDescField(
                    SQLHDESC    DescriptorHandle,
                    SQLSMALLINT RecNumber,
```

```
                    SQLSMALLINT FieldIdentifier,
                    SQLPOINTER  ValuePtr,
                    SQLINTEGER  BufferLength);
```

The following table lists the descriptor fields that can be set using this function in DBMaker. In the table, the *G* represents Get, *S* represents Set, and *I* represents Invalid. SQL_DESC_ARRAY_SIZE only supports a value of one.

FieldIdentifier	ARD	APD	IRD	IPD
SQL_DESC_ALLOC_TYPE	G	G	G	G
SQL_DESC_ARRAY_SIZE	G/S	G/S	I	I
SQL_DESC_ARRAY_STATUS_PTR	G/S	G/S	G/S	G/S
SQL_DESC_BIND_OFFSET_PTR	G/S	G/S	I	I
SQL_DESC_BIND_TYPE	G/S	G/S	I	I
SQL_DESC_COUNT	G/S	G/S	G	G/S
SQL_DESC_ROW_PROCESSED_PTR	I	I	G/S	G/S
SQL_DESC_AUTO_UNIQUE_VALUE	I	I	G	I
SQL_DESC_BASE_COLUMN_NAME	I	I	G	I
SQL_DESC_BASE_TABLE_NAME	I	I	G	I
SQL_DESC_CASE_SENSITIVE	I	I	G	I
SQL_DESC_CATALOG_NAME	I	I	G	I
SQL_DESC_CONCISE_TYPE	G/S	G/S	G	G/S
SQL_DESC_DATA_PTR	G/S	G/S	I	I
SQL_DESC_DATETIME_INTERVAL_ CODE	G/S	G/S	G	G/S
SQL_DESC_DATETIME_INTERVAL_ PRECISION	G/S	G/S	G	G/S
SQL_DESC_DISPLAY_SIZE	I	I	G	I
SQL_DESC_FIXED_PREC_SCALE	I	I	G	G
SQL_DESC_INDICATOR_PTR	G/S	G/S	I	I
SQL_DESC_LABLE	I	I	G	I
SQL_DESC_LENGTH	G/S	G/S	G	G/S
SQL_DESC_LITERAL_PREFIX	I	I	G	I
SQL_DESC_LITERAL_SU FFIX	I	I	G	I
SQL_DESC_LOCAL_TYPE_NAME	I	I	G	G
SQL_DESC_NAME	I	I	G	G

©Copyright 1995-2003 CASEMaker Inc.

FieldIdentifier	ARD	APD	IRD	IPD
SQL_DESC_NULLABLE	I	I	G	G
SQL_DESC_NUM_PREC_RADIX	G/S	G/S	G	G/S
SQL_DESC_OCTET_LENGTH	G/S	G/S	G	G/S
SQL_DESC_OCTET_LENGTH_PTR	G/S	G/S	I	I
SQL_DESC_PARAMETER_TYPE	I	I	I	G/S
SQL_DESC_PRECISION	G/S	G/S	G	G/S
SQL_DESC_SCALE	G/S	G/S	G	G/S
SQL_DESC_SCHEMA_NAME	I	I	G	I
SQL_DESC_SEARCHABLE	I	I	G	I
SQL_DESC_TABLE_NAME	I	I	G	I
SQL_DESC_TYPE	G/S	G/S	G	G/S
SQL_DESC_TYPE_NAME	I	I	G	G
SQL_DESC_UNNAMED	I	I	G	I
SQL_DESC_UNSIGNED	I	I	G	G
SQL_DESC_UPDATABLE	I	I	G	I

The following code fragment shows some commonly used options in the SQLSetDescField function. To simplify the program, table EMPLOYEE only has two columns, (NAME, SALARY).

➲ **Example**

```
SQLHANDLE   hstmt, ard;
SQLSMALLINT status[2];
SQLUCHAR    name[2][18];
Float       salary[2];
SQLRETCODE  retcode;
int         i;
/* retrieve the ard descriptor */
SQLGetStmtAttr(hstmt, SQL_ATTR_APP_ROW_DESC, &ard, NULL, 0);
/* execute a statement */
SQLExecDirect(hstmt, "SELECT * FROM EMPLOYEE", SQL_NTS);
/* set the row size, status and the binding type */
```

©Copyright 1995-2003 CASEMaker Inc.

```
SQLSetDescField(ard, 0, SQL_DESC_ARRAY_SIZE, 2, 0);
SQLSetDescField(ard, 0, SQL DESC ARRAY STATUS PTR, status, 0);
SQLSetDescField(ard, 0, SQL DESC BIND TYPE, SQL BIND BY COLUMN, 0);
/* set necessary fields to fetch the record */
SQLSetDescField(ard, 1, SQL_DESC_CONCISE_TYPE, SQL_C_CHAR, 0);
SQLSetDescField(ard, 1, SQL DESC LENGTH, 18, 0);
SQLSetDescField(ard, 1, SQL_DESC_DATA_PTR, name, 18);
SQLSetDescField(ard, 2, SQL_DESC_CONCISE_TYPE, SQL_C_FLOAT, 0);
SQLSetDescField(ard, 2, SQL_DESC_DATA_PTR, salary, 0);
/* fetch the records */
retcode = SQLFetch(hstmt);
while(retcode == SQL_SUCCESS) {
        for (i = 0; i <2; i++)
        {
        printf("Employee Name : %s, Salary : %f \n", name[i], salary[i]);
        }
        retcode = SQLFetch(hstmt);
}
…
```

SQLSetDescRec

The SQLSetDescRec function is fully supported in the DBMaker API (ODBC 3.0). The SQLSetDescRec function sets the value of multiple descriptor fields that affect the data type and buffer bound to a column or parameter.

Prototype

SQLSetDescRec:

```
RETCODE SQLSetDescField(
                    SQLHDESC     DescriptorHandle,
                    SQLSMALLINT  RecNumber,
```


©Copyright 1995-2003 CASEMaker Inc.

```
                    SQLSMALLINT  Type,
                    SQLSMALLINT  SubType,
                    SQLINTEGER   Length,
                    SQLSMALLINT  Precision,
                    SQLSMALLINT  Scale,
                    SQLPOINTER   DataPtr,
                    SQLINTEGER * StringLengthPtr,
                    SQLINTEGER * IndicatorPtr);
```

The following code fragment shows how to use SQLSetDescRec function to bind columns. To simplify the program, table EMPLOYEE only has two columns (NAME, SALARY).

➲ **Example**

```
SQLHANDLE   hstmt, ard;
SQLSMALLINT status[2];
SQLUCHAR   name[2][19];
SQLINTEGER nameInd[2];
Float      salary[2];
SQLRETCODE retcode;
int        i;
/* retrieve the ard descriptor */
SQLGetStmtAttr(hstmt, SQL_ATTR_APP_ROW_DESC, &ard, NULL, 0);
/* execute a statement */
SQLExecDirect(hstmt, "SELECT * FROM EMPLOYEE", SQL NTS);
/* set the row size, status and the binding type */
SQLSetDescField(ard, 0, SQL_DESC_ARRAY_SIZE, 2, 0);
SQLSetDescField(ard, 0, SQL_DESC_ARRAY_STATUS_PTR, status, 0);
SQLSetDescField(ard, 0, SQL DESC BIND TYPE, SQL BIND BY COLUMN, 0);
/* set necessary fields to fetch the record */
SQLSetDescRec(ard, 1, SQL_C_CHAR, 0, 0, 0, 0, name, 19, nameInd);
```

©Copyright 1995-2003 CASEMaker Inc.

```
SQLSetDescRec(ard, 2, SQL_C_FLOAT, 0, 0, 0,, salary, 0, NULL);
/* fetch the records */
retcode = SQLFetch(hstmt);
while(retcode == SQL_SUCCESS) {
for (i = 0; i < 2; i++)
        {
        printf("Employee Name : %s, Salary : %f \n", name[i], salary[i]);
        }
        retcode = SQLFetch(hstmt);
}
…
```

SQLSetEnvAttr

The SQLSetEnvAttr function is partially supported in the DBMaker API (ODBC 3.0). The SQLSetEnvAttr function sets attributes for an environment handle.

Prototype

SQLSetEnvAttr:

```
RETCODE SQLSetEnvAttr(
                    SQLHENV     EnvironmentHandle,
                    SQLINTEGER  Attribute,
                    SQLPOINTER  ValuePtr,
                    SQLINTEGER  StringLength);
```

The following table lists the attributes that can be set using this function, and whether they are supported in DBMaker. SQL_ATTR_OUTPUT_NTS only supports SQL_TRUE (always NULL terminated).

Attribute	Supported?
SQL_ATTR_ODBC_VERSION	Y
SQL_ATTR_OUTPUT_NTS	Y
SQL_ATTR_CONNECTION_POOLING	N

©Copyright 1995-2003 CASEMaker Inc.

Attribute	Supported?
SQL_ATTR_CP_MATCH	N

SQLSetStmtAttr

The SQLSetStmtAttr function is partially supported in the DBMaker API (ODBC 3.0). The SQLSetStmtAttr function sets attributes for a statement. This function replaces the SQLSetStatementOption function in ODBC 2.0.

➲ **Prototype**

SQLSetStmtAttr:

```
RETCODE SQLSetStmtAttr(
                    SQLHSTMT    StatementHandle,
                    SQLINTEGER  Attribute,
                    SQLPOINTER  ValuePtr,
                    SQLINTEGER  StringLength);
```

The following table lists the attributes that can be set using this function, and whether they are supported in DBMaker. SQL_ATTR_ASYNC_ENABLE and SQL_ATTR_PARAMS_PROCESSED_PTR only support SQL_ASYNC_ENABLE_OFF. SQL_ATTR_CONCURRENCY only supports SQL_CONCUR_READ_ONLY and SQL_CONCUR_LOCK. SQL_ATTR_CURSOR_SENSITIVITY only supports SQL_UNSPECIFIED. SQL_ATTR_KEYSET_SIZE and SQL_ATTR_MAX_LENGTH only support a value of 0. SQL_ATTR_METADATA_ID only supports SQL_FALSE. SQL_ATTR_NOSCAN only supports SQL_NOSCAN_ON. SQL_ATTR_PARAMSET_SIZE only supports a value of one. SQL_ATTR_PARAMSET_SIZE only supports a value of one. SQL_ATTR_QUERY_TIMEOUT only supports a value of 0. SQL_ATTR_RETRIEVE_DATA only supports SQL_RD_ON. SQL_ATTR_SIMULATE_CURSOR only supports SQL_SC_UNIQUE.

Attribute	Supported?
SQL_ATTR_APPPARAM_DESC	Y

©Copyright 1995-2003 CASEMaker Inc.

Attribute	Supported?
SQL_ATTR_APP_ROW_DESC	Y
SQL_ATTR_ASYNC_ENABLE	Y
SQL_ATTR_CONCURRENCY	Y
SQL_ATTR_CURSOR_SCROLLABLE	Y
SQL_ATTR_CURSOR_SENSITIVITY	Y
SQL_ATTR_CURSOR_TYPE	Y
SQL_ATTR_ENABLE_AUTO_IPD	Y
SQL_ATTR_FETCH_BOOKMARK_PTR	Y
SQL_ATTR_IMP_PARAM_DESC	N
SQL_ATTR_IMP_ROW_DESC	N
SQL_ATTR_KEYSET_SIZE	Y
SQL_ATTR_MAX_LENGTH	Y
SQL_ATTR_MAX_ROWS	Y
SQL_ATTR_METADATA_ID	Y
SQL_ATTR_NOSCAN	Y
SQL_ATTR_PARAM_BIND_OFFSET_PTR	Y
SQL_ATTR_PARAM_BIND_TYPE	Y
SQL_ATTR_PARAM_OPERATION_PTR	Y
SQL_ATTR_PARAM_STATUS_PTR	Y
SQL_ATTR_PARAMS_PROCESSED_PTR	Y
SQL_ATTR_PARAMSET_SIZE	Y
SQL_ATTR_QUERY_TIMEOUT	Y
SQL_ATTR_RETRIEVE_DATA	Y
SQL_ATTR_ROW_ARRAY_SIZE	Y
SQL_ATTR_ROW_BIND_OFFSET_PTR	Y
SQL_ATTR_ROW_BIND_TYPE	Y
SQL_ATTR_ROW_NUMBER	N
SQL_ATTR_ROW_OPERATION_PTR	Y
SQL_ATTR_ROW_STATUS_PTR	Y
SQL_ATTR_ROWS_FETCHED_PTR	Y
SQL_ATTR_SIMULATE_CURSOR	Y
SQL_ATTR_USE_BOOKMARK	Y

©Copyright 1995-2003 CASEMaker Inc.

9 Unicode Support

DBMaker now supports native unicode data, so users can now store and process data from multiple languages in the database. Data from multiple languages must be passed as unicode data, however.

The data types NCHAR, NVARCHAR, and NCLOB have been provided to support storage of unicode data. Furthermore, a wide array of unicode functions are now supported.

©Copyright 1995-2003 CASEMaker Inc.

9.1 Unicode Encoding Interfaces

Input data passed from ODBC functions can be UTF-16LE or UTF-8 encoded. You may use a connection option to specify the input Unicode encoding rule. The default encoding is UTF-16LE. When you set this option to UTF-8, DBMaker assumes all input strings are UTF-8 encoded, and the Unicode string is output as UTF8. Windows applications and development tools such as Visual Basic use UTF-16LE encoding; if your AP uses UTF8 (mostly Unix applications), then set the connect option by calling the ODBC set connection option in your program.

DBMaker supports two unicode encoding types to input and output string data, UTF-8 and UTF-16. You can call SQLSetConnectAttr with the options SQL_CLI_UCODE_TYPE and its value SQL_CLI_UTYPE_UTF16 or SQL_CLI_UTYPE_UTF8 to set the input/output unicode string encoding type.

The default value for SQL_CLI_UCODE_TYPE is SQL_CLI_UTYPE_UTF16 (UTF-16LE).

Unicode Functions:

The following ODBC Unicode functions are supported by DBMaker.

SQLColAttributeW

SQLColAttributesW

SQLConnectW

SQLDescribeColW

SQLErrorW

SQLExecDirectW

SQLGetConnectAttrW

SQLGetCursorNameW

SQLGetDescFieldW

©Copyright 1995-2003 CASEMaker Inc.

SQLGetDescRecW

SQLGetDiagFieldW

SQLGetDiagRecW

SQLPrepareW

SQLSetConnectAttrW

SQLSetCursorNameW

SQLSetDescNameW

SQLSetStmtAttrW

SQLGetStmtAttrW

SQLColumnsW

SQLGetConnectOptionW

SQLGetInfoW

SQLGetTypeInfoW

SQLSetConnectOptionW

SQLSpecialColumnsW

SQLStatisticsW

SQLTablesW

SQLDriverConnectW

SQLForeignKeysW

SQLPrimaryKeysW

SQLProcedureColumnsW

SQLProceduresW

Related Unicode ODBC Types

©Copyright 1995-2003 CASEMaker Inc.

For Unicode support, DBMaker supports the following data types: SQL_C_WCHAR for C, and SQL_WCHAR, SQL_WVARCHAR, and SQL_WLONGVARCHAR for SQL. These data types are supported for related ODBC functions, such as SQLBindCol, SQLBindParameter, SQLGetData, SQLPutData, and so on.

The input parameter/output column's length for SQL_C_WCHAR is specified in bytes. The precision of SQL_C_WCHAR is specified in characters.

©Copyright 1995-2003 CASEMaker Inc.

A Function Sequence Differences

This appendix lists differences in function sequences between the DBMaker ODBC API and the Microsoft ODBC 3.0 API.

©Copyright 1995-2003 CASEMaker Inc.

A.1 SQLRowCount

SQLRowCount can be called in state S1, S2, and S3 successfully. DBMaker will return no error. ODBC 3.0 will return error S1010.

©Copyright 1995-2003 CASEMaker Inc.

A.2 SQLGetCursorName

If you call the SQLGetCursorName function before setting a cursor name with the SQLSetCursorName function, DBMaker will not return error S1015 because DBMaker automatically generates a cursor name after allocating a statement handle.

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.

B Function Property Differences

This appendix lists differences in function properties between the DBMaker ODBC API and the Microsoft ODBC 3.0 API. These differences include useful extended options for connection and statements supported by DBMaker that have slightly different behavior than standard ODBC functions.

©Copyright 1995-2003 CASEMaker Inc.

B.1 SQLPutData

ODBC 3.0 allows the SQLPutData function to send data in parts to a column with CHAR, BINARY, LONG VARCHAR and LONG VARBINARY data types. However, DBMaker restricts the data types that can be used with the SQLPutData function to LONG VARCHAR and LONG VARBINARY.

©Copyright 1995-2003 CASEMaker Inc.

B.2 SQLColumns

DBMaker does not allow you to retrieve information from temporary tables by using the SQLColumns function.

©Copyright 1995-2003 CASEMaker Inc.

B.3 SQLTables

DBMaker does not allow you to retrieve information from temporary tables by using the SQLTables function.

©Copyright 1995-2003 CASEMaker Inc.

B.4 SQLDriverConnect

In the SQLDriverConnect function, the behavior of the prompt flags SQL_DRIVER_PROMPT and SQL_DRIVER_COMPLETE_REQUIRED are the same as the behavior of the prompt flag SQL_DRIVER_COMPLETE. Prompt behavior is controlled by the **fDriverComplete** argument of SQLDriverConnect.

©Copyright 1995-2003 CASEMaker Inc.

B.5 SQLBindParameter

For the SQLBindParameter function, the behavior when **pcbValue** is set to SQL_LEN_DATA_AT_EXEC (length) is the same as the behavior when **pcbValue** is set to SQL_DATA_AT_EXEC. The length value of SQL_LEN_DATA_AT_EXEC will be ignored.

©Copyright 1995-2003 CASEMaker Inc.

B.6 Positioned DELETE/UPDATE

An SQL statement containing a positioned DELETE or positioned UPDATE usually refers to a cursor name. If the cursor specified by the cursor name is not open, DBMaker will detect it and return error 34000 at execution time- not preparation time.

©Copyright 1995-2003 CASEMaker Inc.

B.7 SQLSetConnectOption

In the SQLSetConnectOption function, several extended connection options are proprietary to DBMaker. These options allow you to set some of the advanced connection options available when using a DBMaker data source.

Option	Description	Permitted Values
SQL_TXN_ ISOLATION	Sets the transaction isolation level for the current connection.	SQL_TXN_REPEATABLE_ READ SQL_TXN_READ_ UNCOMMITTED SQL_TXN_SERIALIZEABLE SQL_TXN_FORCE_READ_ UNCOMMITTED
SQL_DB_ MODE	Sets the current connecting database to be a single or a multiple-user version.	SQL_SINGLE SQL_MULTI
SQL_ JOURNAL_ MODE	Turns database journal writing on or off	SQL_JOURNAL_ON SQL_JOURNAL_OFF
SQL_LOCK_T IMEOUT	Sets the number of seconds to wait for the lock before returning to the application.	Number of seconds
SQL_ BACKUP_ MODE	Tells the Database what kind of back up it should be use.	SQL_BACKUP_DATA SQL_BACKUP_BLOB SQL_BACKUP_OFF
SQL_DATE_ INPUT_ FORMAT	Tells the database which date format it will receive.	SQL_DATE_FORMAT_0(mm/dd/yy) SQL_DATE_FORMAT_1(mm-dd/yy) SQL_DATE_FORMAT_2(dd/mon/yy) SQL_DATE_FORMAT_3(dd-mon-yy) SQL_DATE_FORMAT_4(mm/dd/yyyy) SQL_DATE_FORMAT_5(mm-dd-yyyy)

©Copyright 1995-2003 CASEMaker Inc.

Option	Description	Permitted Values
		SQL_DATE_FORMAT_6(yyyy/mm/dd) SQL_DATE_FORMAT_7(yyyy-mm-dd) SQL_DATE_FORMAT_8(dd/mon/yyyy) SQL_DATE_FORMAT_9(dd-mon-yyyy) SQL_DATE_FORMAT_10(dd.mm.yyyy) SQL_DATE_FORMAT_11(yy/mm/dd) SQL_DATE_FORMAT_12(yy-mm-dd) SQL_DATE_IN_DEFAULT
SQL_DATE_OUTPUT_FORMAT	Tells the database which date format it should use when returning date type data.	SQL_DATE_FORMAT_0(mm/dd/yy) SQL_DATE_FORMAT_1(mm-dd/yy) SQL_DATE_FORMAT_2(dd/mon/yy) SQL_DATE_FORMAT_3(dd-mon-yy) SQL_DATE_FORMAT_4(mm/dd/yyyy) SQL_DATE_FORMAT_5(mm-dd-yyyy) SQL_DATE_FORMAT_6(yyyy/mm/dd) SQL_DATE_FORMAT_7(yyyy-mm-dd) SQL_DATE_FORMAT_8(dd/mon/yyyy) SQL_DATE_FORMAT_9(dd-mon-yyyy) SQL_DATE_FORMAT_10(dd.mm.yyyy) SQL_DATE_FORMAT_11(yy/mm/dd) SQL_DATE_FORMAT_12(yy-mm-dd) SQL_DATE_OUT_DEFAULT
SQL_TIME_INPUT_FORMAT	Tells the database which time format it will receive.	SQL_DATE_FORMAT_0(hh:mm:ss.fff) SQL_DATE_FORMAT_1(hh:mm:ss) SQL_DATE_FORMAT_2(hh:mm) SQL_DATE_FORMAT_3(hh) SQL_DATE_FORMAT_4(hh:mm:ss.fff tt) SQL_DATE_FORMAT_5(hh:mm:ss tt) SQL_DATE_FORMAT_6(hh:mm tt) SQL_DATE_FORMAT_7(hh tt) SQL_DATE_FORMAT_8(tt hh:mm:ss.fff) SQL_DATE_FORMAT_9(tt hh:mm:ss) SQL_DATE_FORMAT_10(tt hh:mm) SQL_DATE_FORMAT_11(tt hh) SQL_TIME_IN_DEFAULT

©Copyright 1995-2003 CASEMaker Inc.

Option	Description	Permitted Values
SQL_TIME_OUTPUT_FORMAT	Tells the database which time format it should use when returning time type data.	SQL_DATE_FORMAT_0(hh:mm:ss.fff) SQL_DATE_FORMAT_1(hh:mm:ss) SQL_DATE_FORMAT_2(hh:mm) SQL_DATE_FORMAT_3(hh) SQL_DATE_FORMAT_4(hh:mm:ss.fff tt) SQL_DATE_FORMAT_5(hh:mm:ss tt) SQL_DATE_FORMAT_6(hh:mm tt) SQL_DATE_FORMAT_7(hh tt) SQL_DATE_FORMAT_8(tt hh:mm:ss.fff) SQL_DATE_FORMAT_9(tt hh:mm:ss) SQL_DATE_FORMAT_10(tt hh:mm) SQL_DATE_FORMAT_11(tt hh) SQL_TIME_OUT_DEFAULT
SQL_SYSINFO_CLEAR	Asks the database to clear system information.	None
SQL_CB_MODE	Indicates how a COMMIT/ROLLBACK operation affects cursors in the connection.	SQL_CB_CLOSE SQL_CB_RESERVER SQL_CB_DELETE

©Copyright 1995-2003 CASEMaker Inc.

B.8 SQLGetConnectOption

The SQLGetConnectOption function has several extended connection options that are proprietary to DBMaker. These options allow you to get information on some of the advanced connection options available when using a DBMaker data source.

The extended options are:

Option	Description
SQL_TXN_ISOLATION	Gets the transaction isolation level for the current hdbc.
SQL_DB_MODE	Gets the current database mode.
SQL_JOURNAL_MODE	Gets the current database journal-writing mode.
SQL_LOCK_TIMEOUT	Gets the number of seconds to wait for the lock before returning to the application.
SQL_BACKUP_MODE	Gets the current database back up mode.
SQL_DATE_INPUT_FORMAT	Gets the current input format for date type data.
SQL_DATE_OUTPUT_FORMAT	Gets the current output format for date type data.
SQL_TIME_INPUT_FORMAT	Gets the current input format for time type data.
SQL_TIME_OUTPUT_FORMAT	Gets the current output format for time type data.
SQL_CB_MODE	Gets how a COMMIT/ROLLBACK operation affects cursors in the connection.
SQL_CONNECT_ID	Gets the current connection ID.

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.

C ODBC 3.0 Errors

This appendix lists errors in the Microsoft ODBC 3.0 API.

©Copyright 1995-2003 CASEMaker Inc.

C.1 SQLParamData

If the current state is S8, the driver should return SQL_NEED_DATA, not SQL_SUCCESS as defined by ODBC 3.0, after calling SQLParamData.

©Copyright 1995-2003 CASEMaker Inc.

C.2 SQLPrepare

If the current state is S2, the state should be changed to S3 after calling SQLPrepare if the result set is possibly empty.

If the current state is S3, the state should be changed to S2 after calling SQLPrepare if no result set is created.

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.

D Data Types

A driver maps data source-specific SQL data types to ODBC SQL data types and driver-specific SQL data types. Each SQL data type corresponds to an ODBC C data type. The application can specify the correct C data type with the **fCType** argument in SQLBindCol, SQLGetData, or SQLBindParameter. Before sending data to the data source, the driver converts it from the specified C data type. Before retrieving data from the data source, the driver converts it to the specified C data type.

In the following topics will be covered in this section:

- *ODBC* SQL data types
- *ODBC C* data types
- Default *ODBC C* data types
- Precision, scale, length and display size of SQL data types
- Data type conversions

©Copyright 1995-2003 CASEMaker Inc.

D.1 ODBC SQL Data Types

The following table lists the mapping between ODBC SQL data types (the **fSqlType** column) and the corresponding DBMaker SQL data types (SQL Data Type column). A description of the data types is also listed in the table.

FSqlType	SQL Data Type	Description
SQL_CHAR	CHAR(n)	Character string of fixed length n(1 <= n <= 3992)
SQL_VARCHAR	VARCHAR(n)	Variable-length character string with a maximum length n (1 <= n <= 3992).
SQL_LONGVARCHAR	LONG VARCHAR	Variable length character data with a maximum length of 2 GB.
SQL_DECIMAL	DECIMAL(p,s) 1 DECIMAL(p) DECIMAL	Signed exact, numeric value with precision p and scale s. (1<= p <= 17; 0 <= s <= p) (6<= p <= 17; s = 6) (p = 17; s = 6)
SQL_SMALLINT	SMALLINT	Exact signed numeric value with precision 5 and scale 0 (32,768 <= n <= 32,767).
SQL_INTEGER	INTEGER	Exact signed numeric value with precision 10 and scale 0 (-2^{31} <= n <= 2^{31} -1).
SQL_REAL	FLOAT	Signed, approximate, numeric value with a mantissa of precision 7 (10^{-38} to 10^{38}). Slight differences exist between the DBMaker and ODBC 2.0 definition of this data type.

©Copyright 1995-2003 CASEMaker Inc.

FSqlType	SQL Data Type	Description
SQL_FLOAT	DOUBLE	Signed, approximate, numeric value with a mantissa of precision 15 (10^-308 to 10^308).
SQL_DOUBLE	DOUBLE	Signed, approximate, numeric value with a mantissa of precision 15 (10^-308 to 10^308).
SQL_FILE	FILE	DBMaker specific data type. Stores the data of a FILE column as an external file. p (precision) is the total number of digits and s (scale) is the number of digits right of the decimal point.
SQL_BINARY	BINARY(n)	Binary data of fixed length n (1 <= n <= 3992).
SQL_LONGVARBINARY	LONG VARBINARY	Variable length binary data with a maximum length of 2 GB.
SQL_DATE	DATE	Date data.
SQL_TIME	TIME	Time data.
SQL_TIMESTAMP	TIMESTAMP	Includes both date and time data.
SQL_WCHAR	WCHAR(n)	Unicode character string of fixed string length *n* (1<=*n*<=1996)
SQL_WLONGVARCHAR	WLONGVARCHAR	Unicode variable-length character data with a maximum length of 2GB (1GB character length)
SQL_WVARCHAR	WVARCHAR(n)	Unicode variable-length character string with a maximum string length *n* (1<=*n*<=1996)

Table 22 ODBC SQL DataTypes

©Copyright 1995-2003 CASEMaker Inc.

D.2 ODBC C Data Types

The ODBC C data type is the data type used by an application to store data. The C data type used is specified in the SQLBindCol, SQLGetData, and SQLBindParameter functions with the **fCType** argument.

Valid year values are in the range 1 to 9999. Valid year values are in the range 1 to 9999. Valid day values are in the range 1 – number of days in the month. Valid month values are in the range 1 to 12. Valid day values are in the range 1 – number of days in the month. Valid hour values are in the range 0 to 23. Valid minute values are in the range 0 to 59. Valid second values are in the range 0 to 59. Valid fraction values are in the range 0 to 999,999,999. (For example, 500,000,000 represents a half-second, 1, 000,000 is a thousandth of a second, 1,000 is a millionth of a second (a microsecond), and 1 is a billionth of a second (a nanosecond).)

The following table lists the valid **fCType** values, the ODBC C data type that implements each **fCType** value, and the corresponding C type in Windows and UNIX environments.).

FCType	ODBC C Typedef	C Type	Bytes
SQL_C_CHAR	UCHAR FAR *	unsigned char FAR *	4
SQL_C_SSHORT	SWORD	short int	2
SQL_C_SHORT	SWORD	short int	2
SQL_C_USHORT	UWORD	unsigned short int	2
SQL_C_SLONG	SDWORD	long int	4
SQL_C_LONG	SDWORD	long int	4
SQL_C_ULONG	UDWORD	unsigned long int	4
SQL_C_FLOAT	SFLOAT	float	4
SQL_C_DOUBLE	SDOUBLE	double	8
SQL_C_BINARY	UCHAR FAR *	unsigned char FAR *	4
SQL_C_DATE	DATE_STRUCT	struct tagDATE_STRUCT {	6

©Copyright 1995-2003 CASEMaker Inc.

FCType	ODBC C Typedef	C Type	Bytes
		SWORD year; UWORD month; UWORD day; }	

SQL_C_TIME	TIME_STRUCT	struct tagTIME_STRUCT {
		UWORD hour;
		UWORD minutes;
		UWORD second;
		}
			6

SQL_C_TIMESTAMP	TIMESTAMP_STRUCT	struct tagTIMESTAMP_STRUCT {
		SWORD year;
		UWORD month;
		UWORD day;
		UWORD hour;
		UWORD minutes;
		UWORD second;
		UDWORD fraction;
		}
			16

SQL_C_WCHAR	SQLWCHAR FAR*	unsigned short FAR*	4
SQL_C_BOOKMARK	UDWORD	unsigned long int	4
SQL_C_DEFAULT	See the following section "Default C Data Types"		

©Copyright 1995-2003 CASEMaker Inc.

D.3 Default ODBC C Data Types

If an application specifies **SQL_C_DEFAULT** for the fCType argument in SQLBindCol, SQLGetData, or SQLBindParameter, the driver assumes that the C data type of the output or input buffer corresponds to the SQL data type of the column or parameter to which the buffer is bound. The following table shows the default C data type for each ODBC SQL data type. Slight differences exist between the DBMaker and ODBC 2.0 definition of this data type.

SQL Data Type	Default C Data Type
SQL_CHAR	SQL_C_CHAR
SQL_VARCHAR	SQL_C_CHAR
SQL_LONGVARCHAR	SQL_C_CHAR
SQL_DECIMAL	SQL_C_CHAR
SQL_FILE1	SQL_C_CHAR
SQL_SMALLINT	SQL_C_SSHORT
SQL_INTEGER	SQL_C_SLONG
SQL_REAL	SQL_C_FLOAT
SQL_FLOAT	SQL_C_FLOAT
SQL_DOUBLE	SQL_C_DOUBLE
SQL_BINARY	SQL_C_BINARY
SQL_LONGVARBINARY	SQL_C_BINARY
SQL_DATE	SQL_C_DATE
SQL_TIME	SQL_C_TIME
SQL_TIMESTAMP	SQL_C_TIMESTAMP
SQL_WCHAR	SQL_C_WCHAR
SQL_WLONGVARCHAR	SQL_C_WCHAR
SQL_WVARCHAR	SQL_C_WCHAR

©Copyright 1995-2003 CASEMaker Inc.

D.4 Precision, Scale, Length, and Display Size

SQLColAttributes, SQLColumns, and SQLDescribeCol return the precision, scale, length, and display size of a column in a table. SQLDescribeParam returns the precision or scale of a parameter in an SQL statement. SQLBindParameter sets the precision or scale of a parameter in an SQL statement. SQLGetTypeInfo returns the maximum precision and the minimum and maximum scales of an SQL data type on a data source.

A '—' means the value has no meaning or the corresponding type cannot be determined. (For example, scale is not applicable for dates, and the precision of **SQL_LONGVARCHAR** cannot be determined as it only refers to how many bytes of data are stored.)

The precision of date, time and timestamp are all defined as the same as their largest display size, which are the maximum respective lengths of date, time and timestamp format (dd/mon/yyyy, hh:mm:ss.nnn tt, dd/mon/yyyy hh:mm:ss.nnn tt). The scale of time and timestamp are 3. It means the number of digits in fractional part that could be precisely accepted by DBMaker.

The following table lists the precision, scale, length, and display size for each ODBC SQL type.

©Copyright 1995-2003 CASEMaker Inc.

FSqlType	SQL Data Type	Precision	Scale	Length	Display Size
SQL_CHAR	CHAR(n)	n	—	n	n
SQL_VARCHAR	VARCHAR(n)	n	—	n	n
SQL_LONGVARCHAR	LONG VARCHAR	—	—	—	—
SQL_DECIMAL	DECIMAL(p,s)	p	S	(p+3)/2	p+2
SQL_FILE	FILE	79	—	79	—
SQL_SMALLINT	SMALLINT	5	0	2	6
SQL_INTEGER	INTEGER	10	0	4	11
SQL_REAL	FLOAT	7	—	4	13
SQL_FLOAT	FLOAT	7	—	4	13
SQL_DOUBLE	DOUBLE	15	—	8	22
SQL_BINARY	BINARY(n)	n	—	n	2n
SQL_LONGVARBINARY	LONG VARBINARY	—	—	—	—
SQL_DATE	DATE	11	—	6	11
SQL_TIME	TIME	15	3	6	15
SQL_TIMESTAMP	TIMESTAMP	27	3	16	27
SQL_WCHAR	WCHAR(n)	n	—	2n	2n
SQL_WLONGVARCHAR	WLONGVARCHAR	n	—	2n	2n
SQL_WVARCHAR	WVARCHAR(n)	—	—	—	—

©Copyright 1995-2003 CASEMaker Inc.

D.5 Data Type Conversions

Two sections are used to explain data type conversion; one is for SQL to C data conversion, the other is for C to SQL data conversion. In each section, we also illustrate the examples to show the results of specified data type conversion.

SQL to C Data Conversion

Before retrieving data with SQLFetch, you should specify the data type to which the retrieved data is converted in the argument fCType of SQLBindCol. Finally, the driver will store the data in the location pointed to by the **rgbValue** argument in SQLBindCol. SQLGetData has a similar situation. The table following lists the data type conversions, from SQL to C data type, which are provided by DBMaker.

©Copyright 1995-2003 CASEMaker Inc.

SQL Data Type \ C Data Type	SQL_C_CHAR	SQL_C_SSHORT	SQL_C_SHORT	SQL_C_SLONG	SQL_C_LONG	SQL_C_FLOAT	SQL_C_DOUBLE	SQL_C_BINARY	SQL_C_DATE	SQL_C_TIME	SQL_C_TIMESTAMP	SQL_C_FILE	SQL_C_WCHAR
SQL_CHAR	●							○					○
SQL_VARCHAR	●							○					○
SQL_LONGVARCHAR	●							○				○	○
SQL_DECIMAL	●	○	○	○	○	○	○						○
SQL_FILE	●											○	○
SQL_SMALLINT	○	○	●	○	○	○	○						○
SQL_INTEGER	○	○	○	○	●	○	○						○
SQL_REAL	○	○	○	○	○	●	○						○
SQL_FLOAT	○	○	○	○	○	●	○						○
SQL_DOUBLE	○	○	○	○	○	○	●						○
SQL_BINARY	○							●					○
SQL_LONGVARBINARY	○							●				○	○
SQL_DATE	○							○	●		○		○
SQL_TIME	○							○		●	○		○
SQL_TIMESTAMP	○							○	○	○	●		○
SQL_WCHAR	○							○					●
SQL_WLONGVARCHAR	○							○				○	●
SQL_WVARCHAR	○							○					●

● - default conversion
○ - supported conversion

The table on the following three pages illustrates how DBMaker converts SQL data to C data. The output data and returned code are according to the date output format set by the user. If the output format for date is yyyy-mm-dd, then the size of the user buffer must be at least 11 bytes, otherwise an ERROR will be returned. The output data and returned code are according to the date output format set by the user. If the output format for the date is mm-dd-yy, then the size of the user buffer must be at

©Copyright 1995-2003 CASEMaker Inc.

least 9 bytes, otherwise an ERROR will be returned. The output data and returned code are according to the time output format set by the user. If the time output format specifies a fractional part, ex: hh:mm:ss.fff, and the user buffer is not large enough to put the fractional digits, then a warning message of data truncated will be returned.

If the time output format contains a field for seconds, ex: hh:mm:ss, but the user buffer is not large enough to carry the second value, ERROR will be returned. Suppose a user specifies the time output format as hh tt; even though minute and second data is lost, no error will be returned because user only wants the information for hour. The rule for timestamp format combines both the date format rules and time format rules above. So, if the date output format is mm/dd/yy and time output format is hh:mm:ss, and then the buffer size must be at least 18 (8 + 1 + 8 + 1), otherwise an error will be returned.

Suppose a column type is FILE, use SQL_C_CHAR to bind this column will fetch the contents of this file into the user buffer, ex: abcdefg is the contents of a file called 'homework', then abcdefg will be put into the user buffer. The same example as 7, but if you use SQL_C_FILE to bind this column it will fetch the file contents into the new file (ex: student) which is specified by the user buffer.

SQL Data Type	SQL Data Value	C Data Type	C Len	C Data Value	Sqlstate
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0	N/A
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0	01004
SQL_CHAR	abcdef	SQL_C_BINARY	6	abcdef	N/A
SQL_CHAR	abcdef	SQL_C_BINARY	5	abcde	01004
SQL_VARCHAR	same as SQL_CHAR				
SQL_LONGVARCHAR	same as SQL_CHAR				

©Copyright 1995-2003 CASEMaker Inc.

SQL Data Type	SQL Data Value	C Data Type	C Len	C Data Value	Sqlstate
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0	N/A
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	4	—	22003
SQL_DECIMAL	1234.56	SQL_C_FLOAT	—	1234.56	N/A
SQL_DECIMAL	1234.56	SQL_C_SHORT	—	1234	01004
SQL_SMALLINT	7890	SQL_C_CHAR	5	7890\0	N/A
SQL_SMALLINT	7890	SQL_C_CHAR	4	—	22003
SQL_SMALLINT	7890	SQL_C_LONG	—	7890	N/A
SQL_INTEGER	65000	SQL_C_LONG	—	65000	N/A
SQL_INTEGER	65000	SQL_C_SSHORT	—	—	22003
SQL_DOUBLE	1.2345678	SQL_C_DOUBLE	—	1.2345678	N/A
SQL_DOUBLE	1.2345678	SQL_C_FLOAT	—	1.234567	N/A
SQL_DOUBLE	1.2345678	SQL_C_SHORT	—	1	01004
SQL_FLOAT	Same as SQL_DOUBLE				
SQL_REAL	Same as SQL_DOUBLE				
SQL_DATE	1995-11-29	SQL_C_CHAR	11	1995-11-29\0	N/A
SQL_DATE	1995-11-29	SQL_C_CHAR	10	—	22003
SQL_DATE	1995-11-	SQL_C_CH	10	11-29-95\0	N/A

©Copyright 1995-2003 CASEMaker Inc.

SQL Data Type	SQL Data Value	C Data Type	C Len	C Data Value	Sqlstate
	29	AR			
SQL_TIME	22:11:33.32	SQL_C_CHAR	19	22:11:33.320\0	N/A
SQL_TIME	22:11:33.32	SQL_C_CHAR	12	22:11:33.32\0	01004
SQL_TIME	22:11:33.32	SQL_C_CHAR	11	22:11:33.3\0	01004
SQL_TIME	22:11:33.32	SQL_C_CHAR	10	22:11:33\0	01004
SQL_TIME	22:11:33.32	SQL_C_CHAR	9	22:11:33\0	01004
SQL_TIME	22:11:33.32	SQL_C_CHAR	8	—	22003
SQL_TIME	22:11:33.32	SQL_C_CHAR		10 PM\0	N/A
SQL_TIMESTAMP	1995-11-29 23:54:38.234	SQL_C_CHAR	24	1995-11-29 23:54:38.234\0	N/A
SQL_TIMESTAMP	1995-11-29 23:54:38.234	SQL_C_CHAR	22	1995-11-29 23:54:38.2\0	01004
SQL_TIMESTAMP	1995-11-29 23:54:38.234	SQL_C_CHAR	19	—	22003
SQL_TIMESTAMP	1995-11-29 23:54:38.234	SQL_C_CHAR	19	11/29/95 23:54:38\0	N/A
SQL_BINARY	ABCDEFG	SQL_C_CHAR	15	65666768696A6B\0	N/A

©Copyright 1995-2003 CASEMaker Inc.

SQL Data Type	SQL Data Value	C Data Type	C Len	C Data Value	Sqlstate
SQL_BINARY	ABCDEFG	SQL_C_CHAR	14	65666768696A6\0	01004
SQL_BINARY	ABCDEFG	SQL_C_BINARY	7	ABCDEFG	N/A
SQL_BINARY	ABCDEFG	SQL_C_BINARY	6	ABCDEF	01004
SQL_LONGVARBINARY	same as SQL_BINARY				
SQL_FILE	abcdefg	SQL_C_CHAR	8	abcdefg\0	N/A
SQL_FILE	abcdefg	SQL_C_FILE	8	student\0	N/A

C to SQL Data Conversion

When an application calls SQLExecute or SQLExecDirect, the driver retrieves the data for any parameters bound with SQLBindParameter from storage locations in the application. If necessary, before the driver sends data to the data source, it converts the data from the data type specified by the fCType argument in SQLBindParameter to the data type specified by the fSqlType argument in SQLBindParameter. For data-at-execution parameters, the application sends the parameter with SQLPutData. The following table shows the supported conversions from ODBC C data types to ODBC SQL data types by DBMaker.

©Copyright 1995-2003 CASEMaker Inc.

C Data Type \ SQL Data Type	SQL_CHAR	SQL_VARCHAR	SQL_LONGVARCHAR	SQL_DECIMAL	SQL_FILE	SQL_SMALLINT	SQL_INTEGER	SQL_REAL	SQL_FLOAT	SQL_DOUBLE	SQL_BINARY	SQL_LONGVARBINARY	SQL_DATE	SQL_TIME	SQL_TIMESTAMP	SQL_WCHAR	SQL_WVARCHAR	SQL_WLONGVARCHAR
SQL_C_CHAR	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○	○	○	○
SQL_C_SSHORT				○		○	○	○	○	○								
SQL_C_SHORT[1]				○		●	○	○	○	○								
SQL_C_SLONG				○		○	○	○	○	○								
SQL_C_LONG[1]				○		○	●	●	○	○								
SQL_C_FLOAT				○		○	○	○	○	○								
SQL_C_DOUBLE				○		○	○	○	●	●								
SQL_C_BINARY	○	○	○	○		○	○	○	○	○	●	●	○	○	○	○	○	○
SQL_C_DATE	○	○	○										●		○	○	○	○
SQL_C_TIME	○	○	○											●	○	○	○	○
SQL_C_TIMESTAMP	○	○	○										○	○	●	○	○	○
SQL_C_FILE			○									○						
SQL_C_WCHAR	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	●

● - default conversion
○ - supported conversion

The table on the following three pages illustrates how DBMaker converts C data to SQL data. The behavior of this data type is the same as the previous data type in the table. If there is one file in the server site named 'homework' then use SQL_C_CHAR to bind a parameter that will let the specified FILE column link to the file 'homework'. SQL_C_SSHORT, SQL_C_SHORT, SQL_C_SLONG, SQL_C_LONG, SQL_C_FLOAT and SQL_C_DOUBLE are numeric ODBC C data types. SQL_C_DATE cannot be converted to SQL_TIME.

When converting SQL_C_DATE to SQL_TIMESTAMP, the time portion of the timestamp is set to zero. SQL_C_TIME cannot be converted to SQL_DATE. The date part of timestamp will be set to the current date when converting SQL_TIME to SQL_C_TIMESTAMP. If the time portion is not zero, a ‘data truncated’ warning

©Copyright 1995-2003 CASEMaker Inc.

will be returned when converting SQL_C_TIMESTAMP to SQL_DATE. If the fractional part is not zero, a 'data truncated' warning will be returned when converting SQL_C_TIMESTAMP to SQL_TIME.

Since only 3 digits after a decimal in the fractional portion of the timestamp are assured, a warning message will be returned when there are more than 3 digits after decimal. If there is one file called 'homework' on the client site, and its contents are abcdefg, then use SQL_C_FILE to bind a parameter that will insert its contents into the BLOB or FILE column.

C Data Type	C Data Value	SQL Data Type	Col Len	SQL Data	Sqlstate
SQL_C_CHAR	abcdef\0	SQL_CHAR	6	abcdef	N/A
SQL_C_CHAR	abcdef\0	SQL_CHAR	5	abcdef	01004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6,2	1234.56	N/A
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	5,1	1234.5	01004
SQL_C_CHAR	1234.56\0	SQL_DECIMAL	6,3	—	N/A
SQL_C_CHAR	1234\0	SQL_INTEGER	—	1234	N/A
SQL_C_CHAR	1234.56\0	SQL_INTEGER	—	1234	01004
SQL_C_CHAR	12345678 912\0	SQL_INTEGER	—	—	22003
SQL_C_CHAR	abcdef\0	SQL_INTEGER	—	—	22005
SQL_C_CHAR	abcdef\0	SQL_SMALLINT	—	—	22005
SQL_C_CHAR	1234.56\0	SQL_FLOAT	—	1234.56	N/A
SQL_C_CHAR	1234.5678 \0	SQL_FLOAT	—	1234.567 8	N/A
SQL_C_CHAR	1.23456e+ 4\0	SQL_FLOAT	—	1234.56	N/A
SQL_C_CHAR	abcdef\0	SQL_FLOAT	—	—	22005
SQL_C_CHAR		SQL_DOUBLE	same as SQL_FLOAT		
SQL_C_CHAR	66676869 6A6b\0	SQL_BINARY	6	BCDEF	N/A

©Copyright 1995-2003 CASEMaker Inc.

C Data Type	C Data Value	SQL Data Type	Col Len	SQL Data	Sqlstate
SQL_C_CHAR	666768696A6\0	SQL_BINARY	6	BCDEF	N/A
SQL_C_CHAR	666768696A6b\0	SQL_BINARY	5	BCDEF	01004
SQL_C_CHAR	HHKKLLMM\0	SQL_BINARY	6	—	22005
SQL_C_CHAR	1995-11-29\0	SQL_DATE	—	1995-11-29	N/A
SQL_C_CHAR	1995-11-29 00:00:00.00\0	SQL_DATE	—	1995-11-29	N/A
SQL_C_CHAR	1995-11-29 23:54:38.23\0	SQL_DATE	—	1995-11-29	01004
SQL_C_CHAR	1995/22/33	SQL_DATE	—	—	22008
SQL_C_CHAR	17:18:19.123	SQL_TIME	—	17:18:19.123	N/A
SQL_C_CHAR	1995-11-29 17:18:19.123	SQL_TIMESTAMP	—	1995-11-29 17:18:19.123	N/A
SQL_C_CHAR	homework\0	SQL_FILE	9	Abcdefg	N/A
SQL_C_SHORT	7890	SQL_SMALLINT	—	7890	N/A
SQL_C_SSHORT	7890	SQL_VARCHAR	—	—	S1C00
SQL_C_SSHORT	7890	SQL_BINARY	—	—	07006
SQL_C_SSHO	7890	SQL_SMALLINT	—	7890	N/A

©Copyright 1995-2003 CASEMaker Inc.

C Data Type	C Data Value	SQL Data Type	Col Len	SQL Data	Sqlstate
RT					
SQL_C_SSHORT	7890	SQL_INTEGER	—	7890	N/A
SQL_C_USHORT	7890	SQL_INTEGER	—	7890	N/A
SQL_C_LONG	7890	SQL_INTEGER	—	7890	N/A
SQL_C_SLONG	7890	SQL_INTEGER	—	7890	N/A
SQL_C_ULONG	7890	SQL_INTEGER	—	7890	N/A
SQL_C_FLOAT	1234.00	SQL_INTEGER	—	1234	N/A
SQL_C_FLOAT	1234.56	SQL_INTEGER	—	1234	01004
SQL_C_FLOAT	1234567.24	SQL_SMALLINT	—	—	22003
SQL_C_FLOAT	1234.56	SQL_DECIMAL	6,2	1234.56	N/A
SQL_C_FLOAT	1234.56	SQL_DECIMAL	5,1	1234.5	01004
SQL_C_FLOAT	1234.56	SQL_DECIMAL	6,3	—	22003
SQL_C_DOUBLE	same as SQL_C_FLOAT				
SQL_C_BINARY	ABCDEF	SQL_BINARY	6	ABCDEF	N/A
SQL_C_BINARY	ABCDEF	SQL_BINARY	5	ABCDE	01004
SQL_C_DATE	1996,3,8	SQL_DATE	—	1996,3,8	N/A
SQL_C_DATE	1996,3,8	SQL_TIME	—	—	07006
SQL_C_DATE	1996,3,8	SQL_TIMESTAMP	—	1996,3,8,0,0,0,0	N/A
SQL_C_TIME	13,14,15	SQL_DATE	—	—	07006
SQL_C_TIME	13,14,15	SQL_TIME	—	13,14,15	N/A
SQL_C_TIME	13,14,15	SQL_TIMESTAMP	—	1996,3,8,13,14,15	N/A
SQL_C_TIMES	1996,3,8,	SQL_DATE	—	1996,3,8	01004

©Copyright 1995-2003 CASEMaker Inc.

C Data Type	C Data Value	SQL Data Type	Col Len	SQL Data	Sqlstate
TAMP	13,14,15, 160000000				
SQL_C_TIMESTAMP	1996,3,8, 13,14,15, 160000000	SQL_TIME	—	13,14,15	01004
SQL_C_TIMESTAMP	1996,3,8, 13,14,15, 160000000	SQL_TIMESTAMP	—	1996,3,8, 13,14,15, 160000000	N/A
SQL_C_TIMESTAMP	1996,3,8, 13,14,15, 167800000	SQL_TIMESTAMP	—	1996,3,8, 13,14,15, 167000000	01004
SQL_C_FILE	homework\0	SQL_LONGVARCHAR	6	Abcdefg	N/A

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.

E ODBC Log Function

This paragraph lists all configuration options supported by DBMaker for trace logging ODBC functions. You can trace ODBC functions through the Microsoft ODBC Driver Manager on windows platforms, but it does not log functions that are called by Driver Manager itself. In Addition, Microsoft's log function only can be set log on/off to trace all ODBC functions or to not trace them at all, and it can only be used on the windows platform. DBMaker supports a more powerful log function that not only can set log on/off, but also it can specify the ODBC functions that you want to log

You must set a special section named "DM_COMMON_OPTION" in dmconfig.ini, and then you can set various option keywords in this section to enable or disable the ODBC log function.

Note: If you decide not to log ODBC functions later, please remember to remove "DM_COMMON_OPTION" section or set LG_TRACE = 0 in dmconfig.ini to turn off tracing. Otherwise, this log function will record any action you do in data source.

©Copyright 1995-2003 CASEMaker Inc.

The following table lists the options that can be set on the log function in **dmconfig.in***i*, as well as their description and values.

Keyword of Option	Description	Values	Default value
LG_TRACE	Sets the trace flag for the log function.	0-Turn off the ODBC log function. 1-Turn on the ODBC log function.	0
LG_PATH	Sets the path of output log file. If the path is invalid, no log will be output.	Path of output log file.	C:\odbclog.log (for Win32). ./odbclog.log (for Unix).
LG_TIME	Sets whether the time spent by each ODBC function call will be recorded or not.	0-do not log the time spent by each ODBC function call. 1-log the time spent by each ODBC function call.	0
LG_PTFUN	Set the function list to be logged. This list is a string that contains zero or several ODBC function names, separated by commas (,).	Function list string, (e.g. SQLGetDiagRec, SQLCloseCursor, ”).	Not defined (all functions will be logged).

©Copyright 1995-2003 CASEMaker Inc.

Keyword of Option	Description	Values	Default value
LG_NPFUN	Set a function list not to be logged. This list is a string contains zero or several ODBC function names, separated by commas (,). This keyword is valid only if LG_PTFUN is not defined in dmconfig.ini.	Function list string, (e.g. SQLGetDiagRec, SQLCloseCursor, ”).	“” (Empty string, all functions will be logged).

©Copyright 1995-2003 CASEMaker Inc.

©Copyright 1995-2003 CASEMaker Inc.